



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ  
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ  
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

# Σχεδίαση και Υλοποίηση Ροών Εργασίας για Αυτοματοποιημένη Μηχανική Μάθηση στον Κυβερνήτη, με το Kubeflow

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΝΔΡΙΟΠΟΥΛΟΥ Χ. ΚΩΝΣΤΑΝΤΙΝΟΥ

**Επιβλέπων:** Νεκτάριος Κοζύρης  
Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2021

---





# Σχεδίαση και Υλοποίηση Ροών Εργασίας για Αυτοματοποιημένη Μηχανική Μάθηση στον Κυβερνήτη, με το Kubeflow

## ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ ΑΝΔΡΙΟΠΟΥΛΟΥ Χ. ΚΩΝΣΤΑΝΤΙΝΟΥ

**Επιβλέπων:** Νεκτάριος Κοζύρης  
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22η Νοεμβρίου 2021.

(Υπογραφή)

(Υπογραφή)

(Υπογραφή)

.....  
Νεκτάριος Κοζύρης  
Καθηγητής ΕΜΠ

.....  
Γεώργιος Γκούμας  
Αν. Καθηγητής ΕΜΠ

.....  
Ιωάννης Κωνσταντίνου  
Επ. Καθηγητής ΠΘ

.....

**Κωνσταντίνος Χ. Ανδριόπουλος**

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Copyright © - All rights reserved. Με την επιφύλαξη παντός δικαιώματος.  
Κωνσταντίνος Ανδριόπουλος, 2021.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

## Περίληψη

---

Η επίλυση ενός πλήθους από απαιτητικά προβλήματα στις μέρες μας γίνεται με χρήση μηχανικής μάθησης. Μία τέτοια διαδικασία έχει μια βασική δυσκολία, την επιλογή του κατάλληλου αλγορίθμου μηχανικής μάθησης για ένα δεδομένο σύνολο δεδομένων (dataset). Διαφορετικοί αλγόριθμοι μπορεί να έχουν διαφορετικά αποτελέσματα πάνω στο ίδιο dataset, και η διαφορά μπορεί να είναι χαώδης.

Για να λύσει αυτό το πρόβλημα έρχεται η έννοια της Αυτοματοποιημένης Μηχανικής Μάθησης, ή AutoML, που είναι ένα γενικότερο πλαίσιο διαδικασιών και μεθόδων το οποίο παράγει έτοιμα, εκπαιδευμένα μοντέλα με είσοδο κυρίως το σύνολο δεδομένων του χρήστη. Το AutoML αυτοματοποιεί δύσκολες διαδικασίες της μηχανικής μάθησης, όπως την επεξεργασία του συνόλου δεδομένων, την επιλογή του αλγορίθμου αλλά και την εκπαίδευση του αντίστοιχου μοντέλου, διευκολύνοντας με αυτό τον τρόπο την χρήση μηχανικής μάθησης ακόμα και για αυτούς που δεν είναι ειδικοί σε αυτόν τον τομέα.

Υπάρχουν αρκετές βιβλιοθήκες ανοιχτού κώδικα που προσφέρουν λύσεις αυτοματοποιημένης μηχανικής μάθησης εκεί έξω. Μία από αυτές είναι το `auto-sklearn`, το οποίο χρησιμοποιήσαμε στην εργασία μας. Το βασικό πρόβλημα με τέτοιες βιβλιοθήκες είναι ότι, μολονότι μια διαδικασία AutoML συνήθως περιέχει πολλά βήματα που μπορούν να τρέξουν παράλληλα και ανεξάρτητα, αυτές είναι σχεδιασμένες να τρέχουν σε έναν μόνο κόμβο.

Σε αυτήν την εργασία, χρησιμοποιώντας τον πυρήνα του `auto-sklearn` ως βάση, σχεδιάσαμε και υλοποιήσαμε μια διαδικασία αυτοματοποιημένης μηχανικής μάθησης στο `Kubernetes`, το οποίο τρέχει πάνω στον `Kubernetes`, και είναι μια τελευταίας τεχνολογίας πλατφόρμα για εννομήστρωση ροών εργασίας μηχανικής μάθησης. Εκεί, εκμεταλλευτήκαμε τα οφέλη της καταμεμημένης φύσης του `Kubernetes`, κατανέμοντας στο "νέφος" τα βήματα της διαδικασίας AutoML που μπορούσαν να τρέξουν παράλληλα.

## Λέξεις Κλειδιά

Μηχανική Μάθηση, Αυτοματοποιημένη Μηχανική Μάθηση, `Kubernetes`, Δεδομένα, Νέφος, Επιστήμη Δεδομένων



## Abstract

---

Nowadays, a number of demanding real-world problems can be solved with the use of machine learning. One such process has a fundamental difficulty, that is, choosing the most suitable machine learning algorithm for a given dataset. Different algorithms may yield different results, on the same dataset, and their difference can be massive.

AutoML is a solution to that problem. AutoML stands for Automated Machine Learning, and it consists of a plethora of procedures and methods that produce ready-to-use, fully trained models, mainly by receiving the dataset of the user as input. AutoML automates difficult machine learning tasks, such as the preprocessing of the input dataset, choosing an algorithm that is suitable for that dataset, as well as training the corresponding machine learning model. This way, it facilitates the use of machine learning for those that are not necessarily experts in the field.

There is a proliferation of open-source libraries that offer AutoML solutions, out there. One of these libraries is auto-sklearn, which we used in our work. One fundamental problem with such libraries is that, although an AutoML procedure usually contains a large number of steps that can run in parallel, these libraries are designed to run on a single node.

In this diploma thesis, we used auto-sklearn's meta-learning kernel as a base, and we designed and implemented an AutoML process on Kubeflow, which runs on top of Kubernetes and is the state-of-the-art for orchestrating machine learning workflows. There, we leveraged the advantages of Kubernetes' distributed nature by distributing, on the cluster, the steps of the AutoML process that could run in parallel.

## Keywords

Machine Learning, Automated Machine Learning, AutoML, Kubernetes, Kubeflow, auto-sklearn, Cloud, Data Science





*στον αγαπημένο μου πατέρα, του οποίου το άστρο θα με οδηγεί παντοινά*



## Ευχαριστίες

---

Θα ήθελα να εκφράσω την ευγνωμοσύνη μου προς τους ανθρώπους που συνέδραμαν στην εκπόνηση αυτής της διπλωματικής εργασίας, αλλά και στην ευρύτερη ακαδημαϊκή μου πορεία. Καταρχήν, ευχαριστώ πολύ τον επιβλέποντα καθηγητή μου κ. Νεκτάριο Κοζύρη, ο οποίος καλλιέργησε μέσω των διαλέξεων του το ενδιαφέρον μου για τα Υπολογιστικά Συστήματα. Επίσης, ευχαριστώ θερμά τον διδάκτορα Βαγγέλη Κούκη, ο οποίος μου έδωσε την ευκαιρία να εργαστώ μέσα στο περιβάλλον της οικογένειας της Arrikto και ακόμη για την καθοριστική συμβολή του, τόσο στην καλλιέργεια του ενδιαφέροντός μου για ποικίλες πτυχές των υπολογιστικών συστημάτων, όσο και στην διαμόρφωση του τρόπου σκέψης μου ως προς την προσέγγιση ζητημάτων τεχνολογίας λογισμικού. Μέσω της Arrikto, ήρθα σε επαφή με ένα πλήθος εξαιρετών συνεργατών και ανθρώπων. Ευχαριστώ θερμά τον Δημήτρη Πουλόπουλο, για την αμέριστη βοήθεια του ως προς το θεωρητικό κομμάτι της διπλωματικής μου εργασίας. Ακόμα, ευχαριστώ τους Stefano Fioravanzo και Ηλία Κατσακιώρη, οι οποίοι με τις αμέτρητες συμβουλές τους με ωθούσαν αδιάκοπα προς την εξέλιξη μου. Ευχαριστώ επίσης τον καλό μου φίλο Θανάση για την συνεισφορά του στην επιμέλεια αυτής της διπλωματικής. Τέλος, τίποτα από όλα αυτά δε θα ήταν εφικτό χωρίς την αγάπη και τη στήριξη της μητέρας μου, Μαρίας, της αδερφής μου, Δήμητρας, του αγαπημένου μου θείου, Στέφανου, αλλά και του πατέρα μου, Χαράλαμπου, ο οποίος από όταν ήμουν ακόμη βρέφος στιγμάτισε τον χαρακτήρα και την προσωπικότητα μου με έναν τρόπο ξεχωριστό.

Κωνσταντίνος Ανδριόπουλος,  
Νοέμβριος 2021



# Contents

---

<b>Περίληψη</b>	<b>1</b>
<b>Abstract</b>	<b>3</b>
<b>Ευχαριστίες</b>	<b>7</b>
<b>1 Εισαγωγή</b>	<b>19</b>
1.1 Ορισμός του Προβλήματος	19
1.2 Κίνητρο	20
1.3 Σύνοψη Υπαρχόντων Λύσεων	20
1.3.1 Katib	20
1.3.2 Auto-sklearn	21
1.3.3 AutoGluon	22
1.4 Επισκόπηση της Προσέγγισής μας	22
1.5 Δομή της Διπλωματικής Εργασίας	23
<b>2 Υπόβαθρο</b>	<b>25</b>
2.1 Scikit-learn	25
2.1.1 Διοχετεύσεις (Pipelines)	25
2.2 Εικονικοποίηση σε Επίπεδο Λειτουργικού Συστήματος και Περιέκτες (Containers)	26
2.2.1 Επισκόπηση	26
2.2.2 Θεμέλιοι Λίθοι των Περιεκτών	26
2.2.2.1 Ομάδες Ελέγχου (cgroups)	27
2.2.2.2 Χώροι Ονομάτων (Namespaces)	27
2.2.3 Οι Περιέκτες δεν είναι Εικονικές Μηχανές	27
2.2.4 Docker	28
2.2.4.1 Στιγμιότυπα (Images)	28
2.2.4.2 Μητρώα	28
2.3 Κυβερνήτης	29
2.3.1 Επισκόπηση	29
2.3.2 Ελεγκτές και Αντικείμενα	29
2.3.3 Αποθηκευτικός Χώρος	30
2.3.3.1 Λογικοί Δίσκοι	30
2.3.3.2 <i>PersistentVolumes</i> και <i>PersistentVolumeClaims</i>	30
2.4 Kubeflow	30

2.4.1	Επισκόπηση	30
2.4.2	Jupyter Notebooks	31
2.4.3	Διοχετεύσεις (Pipelines)	31
2.4.3.1	Argo	32
2.4.4	Katib	33
2.4.5	MiniKF	34
2.4.5.1	Rok	34
2.5	Kale και Kale-SDK	34
2.5.1	Επισκόπηση	34
2.5.2	Η Κλάση <i>Step</i>	35
2.5.3	Η Κλάση <i>Pipeline</i>	35
2.5.4	Η Γλώσσα Συγκεκριμένου Τομέα του Kale	36
2.5.5	Η Κλάση <i>PythonProcessor</i>	37
2.5.6	Η Κλάση <i>Compiler</i>	37
2.5.7	Μηχανισμός Διαβίβασης Δεδομένων του Kale	38
2.5.8	Εκδόσεις και Στιγμιότυπα Δεδομένων στο Kale	38
2.6	Machine Learning Meta-Data	38
<b>3</b>	<b>Η Μελέτη μας πάνω στον Μηχανισμό Μετα-Μάθησης του <i>Auto-sklearn</i></b>	<b>41</b>
3.1	Επισκόπηση	41
3.2	Διαμορφώσεις Διοχετεύσεων Μηχανικής Μάθησης	42
3.2.1	Επισκόπηση	42
3.2.2	Ο Χώρος Διαμόρφωσης	43
3.2.2.1	Η Συνάρτηση <i>get_configuration_space</i>	43
3.2.3	Ο Κατάλογος Μετα-Δεδομένων	44
3.2.4	Η Κλάση <i>XYDataManager</i>	45
3.2.5	Οι Κλάσεις <i>SimpleRegressionPipeline</i> και <i>SimpleClassificationPipeline</i>	47
3.3	Μετα-Χαρακτηριστικά	47
3.3.1	Επισκόπηση	47
3.3.2	Απλά Μετα-Χαρακτηριστικά	48
3.3.3	Μετα-Χαρακτηριστικά <i>1HotEncoded</i>	49
3.3.4	Συναρτήσεις API για Εξαγωγή Μετα-Χαρακτηριστικών	50
3.3.5	Η Κλάση <i>MetaBase</i>	50
3.4	Η Συνάρτηση <i>suggest_via_metalearning</i>	51
3.5	Βοηθητικές Συναρτήσεις και Κλάσεις	52
3.5.1	Η Κλάση <i>InputValidator</i>	52
<b>4</b>	<b>Η Προσέγγιση μας</b>	<b>53</b>
4.1	Η Συνάρτηση <i>run_automl</i>	55
4.1.1	Το Αντικείμενο <i>Dataset</i>	58
4.2	Ο Ενορχηστρωτής <i>AutoML</i>	58
4.2.1	Η Συνάρτηση Διοχέτευσης <i>automl_orchestrate</i>	59
4.2.2	Το Βήμα <i>GetMetaLearningConfigurations</i>	60

4.2.2.1	Η Συνάρτηση <i>compute_configs</i>	61
4.2.2.2	Η Συνάρτηση <i>_validate_dataset</i>	62
4.2.2.3	Η Συνάρτηση <i>find_metadata_dir</i>	63
4.2.2.4	Η Μέθοδος <i>_submit_configuration_artifact</i>	64
4.2.3	Το Βήμα <i>RunMetaLearningConfigurations</i>	66
4.2.3.1	Η Μέθοδος <i>_run_pipeline</i>	67
4.2.4	Το Βήμα <i>MonitorKFPRuns</i>	68
4.2.5	Το Βήμα <i>GetBestConfiguration</i>	70
4.2.6	Το Βήμα <i>RunKatibExperiment</i>	71
4.2.7	Το Βήμα <i>MonitorKatibExperiment</i>	74
4.3	Οι Ροές Διαμόρφωσης	75
4.3.1	Η Συνάρτηση Διοχέτευσης <i>sklearn_train_predict</i>	76
4.3.2	Το Βήμα <i>RunSKLearnTransformer</i>	77
4.3.3	Το Βήμα <i>TrainSKLearnEstimator</i>	81
4.3.4	Το Βήμα <i>InferSKLearnPredictor</i>	82
4.4	Η Γενεαλογία MLMD και η Παρακολούθηση Κατάστασης των Πειραμάτων AutoML του Kale	85
4.4.1	Artifacts του Kale	85
4.4.1.1	Η Βασική Κλάση <i>MLMDArtifact</i>	85
4.4.1.2	Το Artifact <i>AutoMLConfiguration</i>	88
4.4.1.3	Άλλα Artifacts	89
4.4.2	Το Αντικείμενο <i>AutoMLEXperiment</i>	89
4.4.2.1	Η Μέθοδος <i>list_configurations</i>	91
4.4.2.2	Η Μέθοδος <i>summary</i>	92
<b>5</b>	<b>Αξιολόγηση</b>	<b>101</b>
5.1	Εργαλεία, Μεθοδολογία και Περιβάλλον	101
5.2	Αποτελέσματα	102
<b>6</b>	<b>Συμπερασματικά Σχόλια</b>	<b>105</b>
6.1	Μια Ανακεφαλαίωση του Μηχανισμού μας	105
6.2	Μελλοντικό Έργο	106
<b>7</b>	<b>Introduction</b>	<b>107</b>
7.1	Problem Statement	107
7.2	Motivation	108
7.3	Summary of Existing Solutions	108
7.3.1	Katib	108
7.3.2	Auto-sklearn	109
7.3.3	AutoGluon	109
7.4	Overview of Our Approach	109
7.5	Thesis Structure	110

<b>8 Background</b>	<b>113</b>
8.1 Scikit-learn	113
8.1.1 Pipelines	113
8.2 OS-Level Virtualization and Containers	114
8.2.1 Overview	114
8.2.2 Building Blocks of Containers	114
8.2.2.1 cgroups	114
8.2.2.2 Namespaces	115
8.2.3 Containers are not VMs	115
8.2.4 Docker	115
8.2.4.1 Images	115
8.2.4.2 Registries	116
8.3 Kubernetes	116
8.3.1 Overview	116
8.3.2 Controllers and Objects	117
8.3.3 Storage	117
8.3.3.1 Volumes	117
8.3.3.2 <i>PersistentVolumes</i> and <i>PersistentVolumeClaims</i>	117
8.4 Kubeflow	118
8.4.1 Overview	118
8.4.2 Jupyter Notebooks	118
8.4.3 Pipelines	118
8.4.3.1 Argo	119
8.4.4 Katib	120
8.4.5 MiniKF	121
8.4.5.1 Rok	121
8.5 Kale and the Kale-SDK	121
8.5.1 Overview	121
8.5.2 The <i>Step</i> Class	122
8.5.3 The <i>Pipeline</i> Class	122
8.5.4 Kale's Domain Specific Language	123
8.5.5 The <i>PythonProcessor</i> Class	123
8.5.6 The <i>Compiler</i> Class	124
8.5.7 Kale's Data-Passing Mechanism	124
8.5.8 Kale's Data Versioning and Snapshots	124
8.6 Machine Learning Meta-Data	125
<b>9 Our Study of Auto-sklearn's Meta-Learning Mechanism</b>	<b>127</b>
9.1 Overview	127
9.2 Machine Learning Pipeline Configurations	127
9.2.1 Overview	127
9.2.2 The Configuration Space	129
9.2.2.1 The <i>get_configuration_space</i> API Function	129



9.2.3	The <i>Meta-Data Directory</i> . . . . .	130
9.2.4	The <i>XYDataManager</i> Class . . . . .	130
9.2.5	The <i>SimpleRegressionPipeline</i> and <i>SimpleClassificationPipeline</i> Classes . . . . .	133
9.3	Meta-Features . . . . .	133
9.3.1	Overview . . . . .	133
9.3.2	Simple Meta-Features . . . . .	133
9.3.3	1HotEncoded Meta-Features . . . . .	134
9.3.4	API Functions for Meta-Feature Extraction . . . . .	135
9.3.5	The <i>MetaBase</i> Class . . . . .	136
9.4	The <i>suggest_via_metalearning</i> API Function . . . . .	136
9.5	Utility Functions and Classes . . . . .	137
9.5.1	The <i>InputValidator</i> Class . . . . .	137
<b>10</b>	<b>Our Approach</b> . . . . .	<b>139</b>
10.1	The <i>run_automl</i> API Function . . . . .	141
10.1.1	The Dataset Object . . . . .	144
10.2	The AutoML Orchestrator Pipeline . . . . .	144
10.2.1	The <i>automl_orchestrate</i> Pipeline Function . . . . .	145
10.2.2	The <i>GetMetaLearningConfigurations</i> Step . . . . .	146
10.2.2.1	The <i>compute_configs</i> Function . . . . .	147
10.2.2.2	The <i>_validate_dataset</i> Function . . . . .	150
10.2.2.3	The <i>find_metadata_dir</i> Function . . . . .	151
10.2.2.4	The <i>_submit_configuration_artifact</i> Method . . . . .	152
10.2.3	The <i>RunMetaLearningConfigurations</i> Step . . . . .	154
10.2.3.1	The <i>_run_pipeline</i> Method . . . . .	157
10.2.4	The <i>MonitorKFPRuns</i> Step . . . . .	158
10.2.5	The <i>GetBestConfiguration</i> Step . . . . .	160
10.2.6	The <i>RunKatibExperiment</i> Step . . . . .	162
10.2.7	The <i>MonitorKatibExperiment</i> Step . . . . .	165
10.3	The Configuration Run . . . . .	166
10.3.1	The <i>sklearn_train_predict</i> Pipeline Function . . . . .	168
10.3.2	The <i>RunSKLearnTransformer</i> Step . . . . .	169
10.3.3	The <i>TrainSKLearnEstimator</i> Step . . . . .	172
10.3.4	The <i>InferSKLearnPredictor</i> Step . . . . .	174
10.4	The MLMD Lineage and Status Tracking of Kale-AutoML Experiments . . . . .	177
10.4.1	Kale Artifacts . . . . .	178
10.4.1.1	The <i>MLMDArtifact</i> Base Class . . . . .	178
10.4.1.2	The <i>AutoMLConfiguration</i> Artifact . . . . .	180
10.4.1.3	Other Artifacts . . . . .	181
10.4.2	The <i>AutoMLExperiment</i> Object . . . . .	182
10.4.2.1	The <i>list_configurations</i> Method . . . . .	184
10.4.2.2	The <i>summary</i> Method . . . . .	185

<b>11 Evaluation</b>	<b>193</b>
11.1 Tools, Methodology and Environment . . . . .	193
11.2 Results . . . . .	194
<b>12 Concluding Remarks</b>	<b>197</b>
12.1 A Recap of our Mechanism . . . . .	197
12.2 Future Work . . . . .	198
<b>Βιβλιογραφία</b>	<b>201</b>

## List of Illustrations

---

1.1 Το Auto-sklearn σε μία εικόνα . . . . .	21
2.1 Το γράφημα χρόνου εκτέλεσης μιας διοχέτευσης στο KFP UI . . . . .	32
2.2 Παράδειγμα γραφήματος του Katib UI που δείχνει το επίπεδο επαλήθευσης και εκπαιδευτικής ακρίβειας για διάφορους συνδυασμούς τιμών υπερπαραμέτρων	33
2.3 Το Λογότυπο του Kale . . . . .	34
2.4 Μια Επισκόπηση του MLMD . . . . .	39
4.1 Παράδειγμα μίας διοχέτευσης Ενορχηστρωτή AutoML από την διεπαφή χρήστη του KFP . . . . .	59
4.2 Παράδειγμα ολοκληρωμένης Ροής Διαμόρφωσης στην διεπαφή χρήστη του KFP	76
4.3 Παράδειγμα της εξόδου της μεθόδου <i>list_configurations</i> . . . . .	93
4.4 Παράδειγμα κλήσης της μεθόδου <i>summary</i> μέσα σε ένα κελί Jupyter Notebook	98
4.5 Η σελίδα της διεπαφής χρήστη του KFP που αντιστοιχεί σε μια Ροή Διαμόρφωσης	99
7.1 Auto-sklearn in one image . . . . .	109
8.1 The runtime execution graph of a pipeline in the KFP UI . . . . .	119
8.2 Example graph from the Katib UI, showing the level of validation and train accuracy for various combinations of hyperparameter values . . . . .	120
8.3 The Kale Logo . . . . .	121
8.4 An Overview of MLMD . . . . .	126
10.1 Example AutoML Orchestrator pipeline in the KFP UI . . . . .	145
10.2 Example of completed Configuration Run in the KFP UI . . . . .	167
10.3 Example output of <i>list_configurations</i> method . . . . .	185
10.4 Example output of <i>summary</i> method when executed in a Jupyter Notebook cell . . . . .	191
10.5 The KFP UI page for a Configuration Run . . . . .	192



## List of Tables

---

5.1 Τα σεντ δεδομένων που χρησιμοποιήθηκαν για την αξιολόγηση του AutoML μηχανισμού μας . . . . .	102
5.2 Οι παράμετροι εισόδου για τα αντικείμενα <i>AutoSklearnClassifier</i> και <i>AutoSklearnRegressor</i> που χρησιμοποιήσαμε στα πειράματά μας με το auto-sklearn . . .	102
5.3 Τα αποτελέσματα μετρικής για όλα τα πειράματα. . . . .	102
11.1 The datasets that were used to evaluate our AutoML mechanism . . . . .	194
11.2 The input parameters for <i>AutoSklearnClassifier</i> and <i>AutoSklearnRegressor</i> in our auto-sklearn experiments . . . . .	194
11.3 Metric score measurements for all experiments. . . . .	194



## Chapter 1

# Εισαγωγή

---

Σε αυτό το πρώτο κεφάλαιο, περιγράφουμε συνοπτικά το αντικείμενο της δουλειάς μας. Προσφέρουμε μία σύντομη επισκόπηση της δουλειάς και αποτυπώνουμε το κενό που υπάρχει. Έπειτα, εξετάζουμε τις υπάρχουσες προσεγγίσεις, επισημαίνοντας τις προσφορές και τα μειονεκτήματά τους. Συνεχίζοντας, δίνουμε μία βασική σύνοψη του μηχανισμού που χτίσαμε. Τέλος, παρουσιάζουμε τη δομή αυτής της πτυχιακής εργασίας.

## 1.1 Ορισμός του Προβλήματος

Η κυκλοφορία ενός προϊόντος μηχανικής μάθησης απαιτεί μεγάλους όγκους δεδομένων και επαρκή θεωρητική και πρακτική γνώση τεχνικών μηχανικής μάθησης και αρχιτεκτονικής μοντέλων που μπορούν να εφαρμοστούν σε διαφορετικές εργασίες. Για αυτό το λόγο, η δημιουργία και η διανομή μοντέλων μηχανικής μάθησης που δουλεύουν καλά είναι δύσκολη αποστολή.

Στον πυρήνα του, κάθε επιστήμονας δεδομένων πρέπει να επιλύσει αυτά τα θεμελιώδη προβλήματα της επιλογής αλγορίθμου μηχανικής μάθησης για ένα ορισμένο σετ δεδομένων, εάν και πώς να προ-επεξεργαστεί τα χαρακτηριστικά του, και ποιες τιμές να επιλέξει για τις υπερ-παραμέτρους του. Η εύρεση μιας καλής λύσης για αυτά τα προβλήματα απαιτεί χρόνο και υπολογιστικούς πόρους, αφού θα πρέπει κανείς να πειραματιστεί με έναν αριθμό τεχνικών προ-επεξεργασίας χαρακτηριστικών, αρχιτεκτονικών μοντέλων και τιμών υπερ-παραμέτρων ώστε να βρει ένα συνδυασμό που λειτουργεί επαρκώς ικανοποιητικά. Αυτό είναι το πρόβλημα του *AutoML*, δηλαδή: *η εύρεση ενός μοντέλου που παράγει ακριβείς προβλέψεις δοκιμαστικού σετ για ένα νέο σετ δεδομένων εντός συγκεκριμένου χρονοδιαγράμματος και υπολογιστικού προϋπολογισμού.*

Ένας αριθμός πλαισίων *AutoML*, που αυτοματοποιούν τη διεργασία της *Συνδυασμένης Επιλογής Αλγορίθμου και Ρύθμισης Υπερ-παραμέτρων (Combined Algorithm Selection and Hyper-parameter tuning, CASH)*, προσφέρουν ικανοποιητικές λύσεις σε αυτό το πρόβλημα. Μεταξύ τους, η βιβλιοθήκη *auto-sklearn* [1] της Python που χρησιμοποιήσαμε στην υλοποίηση του μηχανισμού μας και την οποία θα εκθέσουμε αναλυτικά στο κεφάλαιο 3. Ωστόσο, **υπάρχει ένας βασικός περιορισμός σε αυτά τα πλαίσια** και αυτός είναι το γεγονός ότι έχουν χτιστεί πάνω σε κεντροποιημένες βιβλιοθήκες μηχανικής μάθησης (π.χ. η *auto-sklearn* έχει χτιστεί πάνω στην *scikit-learn* [2]) που είναι σχεδιασμένες να δουλεύουν σε ένα μόνο μηχανήμα και ως εκ τούτου τα βήματα μιας διεργασίας *AutoML* που μπορούν να

παραλληλοποιηθούν καταλήγουν να εκτελούνται σειριακά στο μηχάνημα που τα φιλοξενεί. Συνεπώς, τέτοια πλαίσια **δεν είναι εύκολα κλιμακώσιμα**.

Από την άλλη μεριά, πλατφόρμες μηχανικής μάθησης γηγενείς στο νέφος, όπως το *Kubeflow* [3] που θα είναι το κύριο σημείο εστίασης αυτής της εργασίας, **προσφέρουν εννορχήστρωση και κλιμακωσιμότητα για φόρτους εργασίας μηχανικής μάθησης αλλά υστερούν σε εργαλεία AutoML που αξιοποιούν τεχνικές μετα-μάθησης** ώστε να παράξουν ικανοποιητικά μοντέλα.

Όπως συνεπάγεται από τα παραπάνω, **υπάρχει ένα κενό μεταξύ των πλαισίων AutoML και των γηγενών στο νέφος πλατφορμών** που αποτρέπει τους επιστήμονες δεδομένων από το να αξιοποιήσουν τα πλεονεκτήματα και των δύο.

## 1.2 Κίνητρο

Σε κάποιες περιπτώσεις προβλημάτων, η συγγραφή ενός προγράμματος που αντιμετωπίζει το πρόβλημα σε ικανοποιητικό επίπεδο μπορεί να αποδειχθεί δύσκολη για τους ανθρώπους. Η ανάλυση και η λύση του προβλήματος μπορεί να αποδειχθούν μη πρακτικές ή ακόμα και αδύνατες. Σε αυτές τις περιπτώσεις, η μηχανική μάθηση είναι πιθανότατα η καλύτερη επιλογή. Τα μοντέλα μηχανικής μάθησης μπορούν να «ταϊστούν» με μεγάλους όγκους δεδομένων, να αναγνωρίσουν μοτίβα σε αυτά και να επιλύσουν το πρόβλημα αποτελεσματικά.

Οι επιστήμονες δεδομένων είναι αντιμέτωποι με το δύσκολο πρόβλημα του να φέρουν τα δεδομένα σε καλή κατάσταση και έπειτα **να βρουν μία κατάλληλη διαμόρφωση μοντέλου μηχανικής μάθησης για την επίλυση ενός δεδομένου προβλήματος**. Επιπλέον, πρέπει να εκτελούν όλα τα προαναφερθέντα εγκαίρως και αποδοτικά από άποψη κόστους και να μπορούν να παράγουν μοντέλα που είναι ακριβή και εύκολα στην κλιμάκωση και τη διανομή.

Οι επιστήμονες δεδομένων είναι το κοινό που στοχεύουμε με αυτή τη δουλειά. Ο κύριος στόχος μας είναι να απλουστεύσουμε τις ζωές τους προσφέροντας λύση στα προαναφερθέντα προβλήματα.

## 1.3 Σύνοψη Υπαρχόντων Λύσεων

Σε αυτή την ενότητα, παρουσιάζουμε συνοπτικά τις πιο αξιοσημείωτες λύσεις στο πεδίο του AutoML.

### 1.3.1 Katib

Το Katib [4] είναι το εργαλείο του Kubeflow για τη βελτιστοποίηση υπερ-παραμέτρων. Η κύρια ιδέα πίσω από αυτό είναι ο ορισμός της διαδικασίας εκπαίδευσης εντός ενός container και η εκτέλεση αυτής της λογικής πολλαπλές φορές, αλλάζοντας τις υπερ-παραμέτρους του υπό τη μορφή container μοντέλου κάθε φορά, μέσω των ορισμάτων εισόδου της εντολής του σημείου εισόδου του container, έως ότου να καταλήξει σε ένα ικανοποιητικό σει τιμών υπερ-παραμέτρων.



Το Katib υποστηρίζει έναν αριθμό αλγορίθμων αναζήτησης τους οποίους χρησιμοποιεί για να βρει σει υπερ-παραμέτρων που ικανοποιούν τις απαιτήσεις που θέτει ο χρήστης σχετικά με την τελική απόδοση του μοντέλου. Μερικοί από αυτούς τους αλγορίθμους είναι: *Grid Search* και *Μπεϋζιανή Βελτιστοποίηση*. Επιπλέον, το Katib υποστηρίζει αλγορίθμους αναζήτησης νευρωνικής αρχιτεκτονικής *ENAS* και *DARTS* για την εύρεση αρχιτεκτονικών νευρωνικών δικτύων που είναι βέβαιο ότι θα λειτουργούν καλά για μια δεδομένη εργασία. Για μια πιο λεπτομερή επισκόπηση όλων των διαφορετικών αλγορίθμων αναζήτησης που προσφέρει το Katib, ανατρέξτε στην αντίστοιχη ενότητα της επίσημης τεκμηρίωσης του Katib [5].

**Ένα βασικό μειονέκτημα** του Katib είναι το γεγονός ότι ο χρήστης πρέπει να επιλέξει την αρχιτεκτονική/τον τύπο του μοντέλου και να παρέχει την υλοποίηση του μοντέλου για την οποία το Katib θα αναζητήσει το χώρο υπερ-παραμέτρων ώστε να βρει ένα υψηλής βαθμολογίας σει υπερ-παραμέτρων. **Η επιλογή του σωστού μοντέλου για ένα συγκεκριμένο σει δεδομένων είναι ήδη ένα δύσκολο εγχείρημα από μόνη της.** Επιπλέον, η βελτιστοποίηση των υπερ-παραμέτρων είναι βέβαιη ότι θα είναι αναποτελεσματική, εάν η αρχιτεκτονική του μοντέλου δεν είναι κατάλληλη για το δοσμένο σει δεδομένων και πρόβλημα.

### 1.3.2 Auto-sklearn

Το **Auto-sklearn** [1] είναι μια βιβλιοθήκη της Python για αυτοματοποιημένη μηχανική μάθηση (AutoML) που **απαλλάσσει ένα χρήστη μηχανικής μάθησης από την επιλογή αρχιτεκτονικής μοντέλου** και τη ρύθμιση υπερ-παραμέτρων. Αξιοποιεί πρόσφατα πλεονεκτήματα στη Μπεϋζιανή βελτιστοποίηση, τη μετα-μάθηση [6] και την κατασκευή ενωμένων μοντέλων και επιτυγχάνει να αντικαθιστά πλήρως οποιονδήποτε εκτιμητή *scikit-learn* ([2]), για επιβλεπόμενες εργασίες μηχανικής μάθησης. Για τους σκοπούς αυτής της διπλωματικής εργασίας, αξιοποιήσαμε το μηχανισμό και τη βάση δεδομένων μετα-μάθησης του *auto-sklearn* ώστε να δημιουργήσουμε μία κατανεμημένη διεργασία AutoML στο Kubeflow.

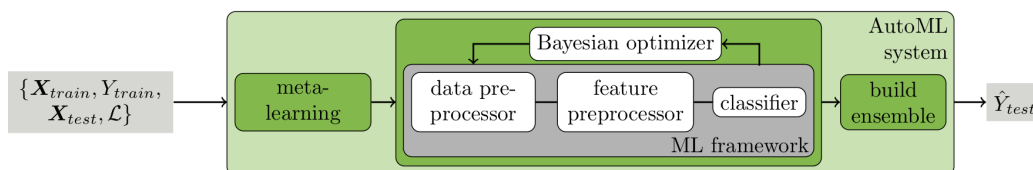


Figure 1.1: Το Auto-sklearn σε μία εικόνα

**Το κύριο μειονέκτημα** του auto-sklearn είναι το γεγονός ότι, παρόλο που ξεκινά έναν αριθμό ανεξάρτητων εργασιών εκπαίδευσης για διαφορετικές διαμορφώσεις μοντέλων ώστε να βρει αυτό με την υψηλότερη βαθμολογία για ένα δοσμένο σει δεδομένων, έχει χτιστεί πάνω στο *scikit-learn* που είναι μια κεντροποιημένη βιβλιοθήκη μηχανικής μάθησης και είναι σχεδιασμένο να τρέχει σε ένα μόνο μηχάνημα.

Σε αυτή την εργασία, μεταφέρουμε τη διαδικασία του auto-sklearn στον *Κυβερνήτη* [7], αξιοποιώντας την κατανεμημένη φύση του, ώστε να εκπαιδεύσουμε μοντέλα μηχανικής μάθησης ως εργασίες παράλληλων διοχετεύσεων. Για μια πιο εμπειριστωμένη αντίληψη του μηχανισμού μετα-μάθησης του *auto-sklearn*, ανατρέξτε στο κεφάλαιο 3.

### 1.3.3 AutoGluon

Το AutoGluon [8] είναι ακόμα μία βιβλιοθήκη της Python γνωστή για εργασίες AutoML. Είναι ουσιαστικά ένα πλαίσιο AutoML ανοιχτού κώδικα το οποίο (παρόμοια με το *auto-sklearn*) απαιτεί λίγες μόνο γραμμές κώδικα Python ώστε να εκπαιδεύσει υψηλής ακρίβειας μοντέλα μηχανικής μάθησης σε ένα μη συμπιεσμένο σετ δεδομένων. Εστιάζει κυρίως σε δομημένα δεδομένα, όπως δεδομένα κειμένου, εικόνας και πινάκων και εκτελεί προηγμένα επεξεργασία δεδομένων, βαθιά μάθηση, και πολυεπίπεδη ένωση μοντέλων ώστε να μεγιστοποιήσει τα αποτελέσματά του.

## 1.4 Επισκόπηση της Προσέγγισής μας

Όπως εξηγήσαμε παραπάνω, υπάρχει ένα κενό μεταξύ εργαλειοθηκών AutoML, όπως το *auto-sklearn*, και γηγενών στο νέφος πλατφορμών, όπως το *Kubeflow*, που αποτρέπει το συνδυασμό των πλεονεκτημάτων και των δύο.

Το Kubeflow [3] είναι μία εξαιρετική πλατφόρμα για την ενορχήστρωση πολύπλοκων ροών εργασιών πάνω στον Κυβερνήτη. Η αυτοεξυπηρετούμενη φύση του το καθιστά εξαιρετικά ελκυστικό για επιστήμονες δεδομένων, καθώς προσφέρει εύκολη πρόσβαση σε προηγμένη ενορχήστρωση καταναμημένων εργασιών, επαναχρησιμοποιησιμότητα μερών, Jupyter Notebooks [9], Pipelines [10], πλούσια UIs και ακόμα περισσότερα.

Στη δουλειά μας, επεκτείνουμε το Kale [11], μια εργαλειοθήκη ενορχήστρωσης διοχετεύσεων για το Kubeflow, ώστε να χρησιμοποιεί το μηχανισμό μεταμάθησης του *auto-sklearn* για να παράγει πλήρως εκπαιδευμένα μοντέλα επιβλεπόμενης μάθησης που είναι βέβαια να λειτουργούν καλά για ένα δοσμένο σετ δεδομένων, αξιοποιώντας ταυτόχρονα μέρη του Kubeflow και την καταναμημένη φύση του Κυβερνήτη.

Η ακόλουθη **αριθμημένη λίστα βημάτων** περιγράφει το μηχανισμό πίσω από τη διεργασία AutoML που χτίσαμε για το Kale, από την οπτική του χρήστη:

1. Ο χρήστης προμηθεύει ένα **σετ δεδομένων** και τον **τύπο του προβλήματος μηχανικής μάθησης** (ταξινόμηση ή παλινδρόμηση) σαν είσοδο στη συνάρτηση `run_automl()` του API του Kale.
2. **Με μία απλή κλήση συνάρτησης, η όλη διεργασία ξεκινά** και το Kale δημιουργεί μία διοχέτευση KubeFlow ώστε να ενορχηστρώσει την όλη διεργασία.
3. **Η συνάρτηση `run_automl` του API επιστρέφει ένα αντικείμενο Python στο χρήστη** για να παρακολουθεί την κατάσταση όλης της διεργασίας AutoML.
4. Χρησιμοποιώντας τον πυρήνα μεταμάθησης του *auto-sklearn*, **η διοχέτευση ενορχηστρωτής υπολογίζει μία λίστα διαμορφώσεων μοντέλων μεταμάθησης** που είναι βέβαιο ότι θα αποδώσουν για το σετ δεδομένων εισαγωγής. Αυτές οι διαμορφώσεις **περιγράφουν πλήρως ολόκληρες διοχετεύσεις μηχανικής μάθησης** (π.χ. προεπεξεργαστές δεδομένων και αρχιτεκτονική μοντέλων).
5. **Για κάθε διαμόρφωση μηχανικής μάθησης, η διοχέτευση ενορχηστρωτής δημιουργεί μία καινούργια διοχέτευση.**

6. **Αυτές οι νέες διοχτετεύσεις εκτελούνται παράλληλα**, προεπεξεργάζοντας το σετ δεδομένων, εκπαιδεύοντας το μοντέλο που προτείνει η αντίστοιχη διαμόρφωση, και παράγοντας βαθμολογίες των δοκιμών **ενώ ο ενορχηστρωτής τις επιβλέπει**.
7. Αφού έχουν όλες ολοκληρωθεί, **ο ενορχηστρωτής συγκεντρώνει τις βαθμολογίες τους και επιλέγει το μοντέλο από τη διοχτέυση με την καλύτερη βαθμολογία**.
8. **Ο ενορχηστρωτής δημιουργεί ένα πείραμα Katib για περαιτέρω βελτιστοποίηση του μοντέλου με την καλύτερη βαθμολογία**.
9. **Το Kale αποθηκεύει το εκπαιδευμένο και βελτιστοποιημένο μοντέλο και λαμβάνει ένα πλήρως αναπαράξιμο στιγμιότυπο (2.5.8) του λογικού δίσκου που το περιέχει** ώστε ο χρήστης να μπορεί αργότερα να έχει πρόσβαση και να το αναπαράξει εύκολα.

**Αυτή η πτυχιακή εργασία επικεντρώνεται κυρίως στα βήματα 3 έως 7 της διεργασίας Kale-AutoML.** Παρ' όλα αυτά, θα προσφέρουμε μία επαρκή ανάλυση του υπόλοιπου μηχανισμού επίσης.

## 1.5 Δομή της Διπλωματικής Εργασίας

Το υπόλοιπο έγγραφο οργανώνεται ως εξής:

- **Στο κεφάλαιο 2** παραθέτουμε το απαραίτητο θεωρητικό υπόβαθρο ώστε ο αναγνώστης να κατανοήσει τη δουλειά μας.
- **Στο κεφάλαιο 3** εκθέτουμε την κατανόηση και γνώση μας γύρω από το μηχανισμό μεταμάθησης του *auto-sklearn*.
- **Στο κεφάλαιο 4** αναλύουμε το σχεδιασμό και την υλοποίηση του μηχανισμού μας.
- **Στο κεφάλαιο 5** αξιολογούμε τη δουλειά μας.
- **Στο κεφάλαιο 6** προσφέρουμε μία σύνοψη των συνεισφορών μας καθώς και πιθανές μελλοντικές κατευθύνσεις του έργου.



## Chapter 2

# Υπόβαθρο

---

Σε αυτό το κεφάλαιο παραθέτουμε το απαραίτητο θεωρητικό υπόβαθρο για την κατανόηση των κεντρικών ιδεών στη συνέχεια της πτυχιακής εργασίας.

## 2.1 Scikit-learn

Η *Scikit-learn* (επίσης γνωστή ως *sklearn*) [2] είναι μία ανοιχτού κώδικα βιβλιοθήκη μηχανικής μάθησης για την Python [12]. Διαθέτει ποικίλους αλγόριθμους κατηγοριοποίησης, παλινδρόμησης και ομαδοποίησης, συμπεριλαμβανομένων μηχανών υποστηρικτικών διανυσμάτων, τυχαίων δασών, ενίσχυσης κλίσεων, ομαδοποίησης κ-μέσων και DBSCAN, και είναι σχεδιασμένη να διαλειτουργεί με τις αριθμητικές και επιστημονικές βιβλιοθήκες της Python *NumPy* [13] και *SciPy* [14].

### 2.1.1 Διοχετεύσεις (Pipelines)

Μία διοχέτευση μηχανικής μάθησης είναι ουσιαστικά το προϊόν της αλυσιδωτής σύνδεσης μιας αλληλουχίας βημάτων που συμπεριλαμβάνονται σε ένα μοντέλο μηχανικής μάθησης. Μπορεί να χρησιμοποιηθεί για να αυτοματοποιήσει μια ροή εργασιών μηχανικής μάθησης. Η διοχέτευση μπορεί να περιλαμβάνει εργασίες όπως:

1. **προεπεξεργασία**
2. **επιλογή χαρακτηριστικών**
3. **κατηγοριοποίησης/παλινδρόμηση**

Επιπλέον, πιο περίπλοκες εφαρμογές ενδέχεται να απαιτούν την ενσωμάτωση άλλων απαραίτητων βημάτων εντός αυτής της διοχέτευσης.

Το δομοστοιχείο *pipeline* της *Scikit-learn* προσφέρει ένα αντικείμενο κλάσης *Pipeline* [15] το οποίο συνδέει τα βήματα **μετασχηματιστής** and **εκτιμητής** της *scikit-learn* σε μία ενιαία διοχέτευση μηχανικής μάθησης. Η κύρια μέθοδος του API της, *fit\_transform*, καλεί διαδοχικά τις *fit\_transform* μεθόδους όλων των βημάτων που υπάρχουν στη διοχέτευση.

Η *Scikit-learn* χρησιμοποιεί τον όρο «μετασχηματιστές» για να αναφερθεί σε αντικείμενα που εφαρμόζουν προεπεξεργασία δεδομένων/χαρακτηριστικών στην *scikit-learn*, και προσφέρουν μία μέθοδο *fit\_transform*, η οποία πιθανότατα καθαρίζει, μειώνει, επεκτείνει ή, γενικά, επεξεργάζεται ένα σύνολο δεδομένων. Από την άλλη, η *scikit-learn* χρησιμοποιεί

τον όρο «εκτιμητές» για να αναφερθεί σε μοντέλα μηχανικής μάθησης που μπορούν να εκπαιδευτούν πάνω σε ένα εισακτέο σύνολο δεδομένων, μέσω μίας μεθόδου *fit*, και είναι ικανά να παράξουν προβλέψεις για νέα, άγνωστα δεδομένα, μέσω μίας μεθόδου *predict*.

Στην περίπτωση ενός *Pipeline* που αποτελείται μόνο από μετασχηματιστές, η μέθοδος *fit\_transform* του *Pipeline* καλεί στην πραγματικότητα την *fit\_transform* του πρώτου μετασχηματιστή, μετά παρέχει την έξοδό της στον επόμενο μετασχηματιστή κ.ο.κ.

Στη γενική περίπτωση ωστόσο, ένα *Pipeline* αποτελείται από μετασχηματιστές και έναν εκτιμητή σαν ένα τελικό βήμα. Σε αυτήν την περίπτωση, η μέθοδος *fit\_transform* του αντικειμένου *Pipeline* καλεί τις μεθόδους *fit\_transform* των βημάτων των μετασχηματιστών διαδοχικά, και μετά εκπαιδεύει τον εκτιμητή στα μετασχηματισμένα δεδομένα καλώντας τη μέθοδο *fit* του βήματος του εκτιμητή.

## 2.2 Εικονικοποίηση σε Επίπεδο Λειτουργικού Συστήματος και Περιέκτες (Containers)

### 2.2.1 Επισκόπηση

Η εικονικοποίηση επιπέδου λειτουργικού συστήματος είναι μία λειτουργία ενός λειτουργικού συστήματος με την οποία υπηρεσίες του πυρήνα επιτρέπουν τη συνύπαρξη πολλαπλών αντικειμένων χώρων χρήση σαν το καθένα να είναι απομονωμένο από τα υπόλοιπα. Υπάρχει ένας ενδιαφέρον τύπος αντικειμένων εικονικοποίησης επιπέδου λειτουργικού συστήματος, τα οποίες είναι γρήγορα, ελαφριά και εύκολα στη χρήση. Αυτά τα αντικείμενα ονομάζονται: περιέκτες (containers).

Οι περιέκτες εμφανίζονται σαν πραγματικά, αυτόνομα μηχανήματα από τη σκοπιά των διεργασιών που εκτελούνται εντός τους. Μπορούν ουσιαστικά να τρέξουν παράλληλα, ενόσω μοιράζονται το λειτουργικό σύστημα του οικοδεσπότη υπολογιστή. Αυτό σημαίνει ότι κάθε περιέκτης χρησιμοποιεί τη διεπαφή κλήσεων συστήματος του λειτουργικού συστήματος και δεν χρειάζεται να εξομοιωθούν ή να τρέξουν σε εικονικό μηχανήμα. Αυτό καθιστά τους περιέκτες πολύ ελαφρούς, αφού απαιτούν λιγότερο επιπλέον κόστος για να εκκινηθούν, εν αντιθέσει με τεχνολογίες πλήρους εικονικοποίησης. Από μία σκοπιά υψηλού επιπέδου, αυτό είναι που τα διαφοροποιεί από τα εικονικά μηχανήματα.

Οι περιέκτες προσφέρουν ένα μηχανισμό λογικού συσκευασμού με τον οποίο οι εφαρμογές μπορούν να είναι αποκομμένες από το περιβάλλον στο οποίο εκτελούνται στην πραγματικότητα. Αυτή η απόζευξη επιτρέπει την εύκολη και συνεπή διανομή εφαρμογών βασισμένων σε περιέκτες, ασχέτως εάν το στοχευμένο περιβάλλον είναι ένα ιδιωτικό κέντρο δεδομένων, το δημόσιο υπολογιστικό νέφος, ή ακόμα και το προσωπικό laptop ενός developer.

### 2.2.2 Θεμέλιοι Λίθοι των Περιεκτών

Από τη σκοπιά μιας ομάδας μηχανικών, ένας περιέκτης είναι μία τυπική μονάδα παράδοσης λογισμικού που διευκολύνει την παραγωγή και διανομή λογισμικού. Ούτως ώστε να αποκτήσουμε μια πιο πλήρη κατανόηση των δυνατοτήτων και των περιορισμών τους, πρέπει

να εξετάσουμε τους μηχανισμούς που λειτουργούν σαν θεμέλιοι λίθοι για τους περιέκτες στο παρασκήνιο, και να μπορέσουμε να δρέψουμε τα οφέλη τους.

### 2.2.2.1 Ομάδες Ελέγχου (cgroups)

Οι ομάδες ελέγχου (cgroups) είναι ουσιαστικά ένας μηχανισμός ιεραρχικής οργάνωσης διεργασιών και κατανομής των πόρων του συστήματος κατά μήκος της ιεραρχίας με έναν ελεγχόμενο και ρυθμιζόμενο τρόπο ([16]).

Χρησιμοποιώντας τις cgroups, ο διαχειριστής του συστήματος μπορεί να αναθέσει ένα σετ ορίων στη χρήση πόρων μιας συλλογής διεργασιών οι οποίες οριοθετούνται από τα ίδια κριτήρια. Η οργάνωση των ομάδων μπορεί να είναι ιεραρχική, υπό την έννοια ότι κάθε ομάδα κληρονομεί τις ρυθμίσεις του γονέα της.

Ο πυρήνας του Linux εκθέτει μια ποικιλία ελεγκτών (υποσυστήματα) μέσω της διεπαφής cgroup που χρησιμοποιούνται για να περιορίσουν τη χρήση πόρων αυτών των ομάδων. Για παράδειγμα, ο ελεγκτής μνήμη περιορίζει τη χρήση μνήμης και ο cpusacct καταγράφει τη χρήση της CPU.

### 2.2.2.2 Χώροι Ονομάτων (Namespaces)

Οι χώροι ονομάτων (namespaces) είναι μια λειτουργία του πυρήνα του Linux που επιτρέπει τη διαίρεση των πόρων του πυρήνα με τέτοιο τρόπο που διαφορετικά σετ διεργασιών έχουν πρόσβαση σε διαφορετικά σετ πόρων. Μερικά παραδείγματα τέτοιων πόρων είναι οι ταυτότητες διεργασιών, τα ονόματα αρχείων και αρχεία σχετιζόμενα με την πρόσβαση στο δίκτυο.

## 2.2.3 Οι Περιέκτες δεν είναι Εικονικές Μηχανές

Ενώ τόσο οι περιέκτες όσο και οι εικονικές μηχανές είναι ουσιαστικά υλοποιήσεις εικονικοποίησης, διαφέρουν σημαντικά.

Ένα κύριο χαρακτηριστικό των εικονικών μηχανημάτων είναι ότι εκτελούν ένα εντελώς ξεχωριστό φιλοξενούμενο λειτουργικό σύστημα και εξομοιώνουν τις συσκευές υλικού του, δηλαδή ουσιαστικά εικονικοποιούν τη στοίβα υλικού. Για παράδειγμα, διαφορετικά εικονικά μηχανήματα, τα οποία εκτελούνται στο ίδιο λειτουργικό σύστημα οικοδεσπότη, έχουν τη δικιά τους, ξεχωριστή εικόνα στο δίσκο. Τα εικονικά μηχανήματα επίσης προσφέρουν αυστηρή ασφάλεια και απομόνωση ανάμεσα στις εργασίες.

Οι περιέκτες, αντιθέτως, μοιράζονται το υποκείμενο λειτουργικό σύστημα. Αντί να εικονικοποιήσουν τη στοίβα υλικού, εικονικοποιούνται στο επίπεδο του λειτουργικού συστήματος, με πολλαπλούς περιέκτες να εκτελούνται απευθείας πάνω στον πυρήνα του λειτουργικού συστήματος. Αυτό σημαίνει ότι οι περιέκτες είναι πολύ πιο ελαφροί. Μοιράζονται τον πυρήνα του λειτουργικού συστήματος, ξεκινούν πολύ γρηγορότερα, και χρησιμοποιούν ένα μέρος μόνο της μνήμης του μηχανήματος, συγκριτικά με την εκκίνηση ενός ολόκληρου λειτουργικού συστήματος, όπως στην περίπτωση των εικονικών μηχανημάτων.

Οι περιέκτες πακετάρονται με ένα ελάχιστο σετ απαραίτητων εξαρτήσεων (βιβλιοθήκες, άλλα αρχεία) και τα στιγμιότυπα των περιεκτών είναι συνήθως (αυτό εξαρτάται από την εφαρμογή που γίνεται περιέκτης) τάξεις μεγέθους μικρότερες από τα στιγμιότυπα εικονικών μηχανημάτων.

## 2.2.4 Docker

Το Docker είναι μια σειρά προϊόντων πλατφόρμα-ως-υπηρεσία που χρησιμοποιούν εικονικοποίηση επιπέδου λειτουργικού συστήματος για να παραδώσουν λογισμικό σε πακέτα. Αυτά τα πακέτα είναι ουσιαστικά *περιέκτες*, ή πιο συγκεκριμένα: στιγμιότυπα περιεκτών, η σημασία των οποίων θα εξηγηθεί ακριβώς από κάτω. Το Docker χρησιμοποιεί τους προαναφερθείς θεμέλιους λίθους για να δημιουργήσει μία διεπαφή που καθιστά ευκολότερο το χειρισμό και την παραμετροποίηση των περιεκτών, όπως επίσης και των εφαρμογών που εκτελούνται εντός τους.

### 2.2.4.1 Στιγμιότυπα (Images)

Οι περιέκτες, σαν τεχνολογία, εμφανίστηκαν για να καλύψουν την ανάγκη για επαναχρησιμοποιησιμότητα και αναπαραγωγιμότητα εφαρμογών λογισμικού. Για να συμβεί αυτό, ομάδες μηχανικής λογισμικού χρειάζονταν έναν τρόπο να παγώσουν την κατάσταση των περιεκτών, ώστε να μπορούν αργότερα να στείλουν αυτούς τους παγωμένους περιέκτες σε χρήστες που θα τρέξουν τις εφαρμογές υπό τη μορφή περιεκτών. Αυτή η παγωμένη έκδοση ενός περιέκτη ονομάζεται στιγμιότυπο του περιέκτη, και είναι μία ιδέα που εισήχθηκε πρώτη φορά από το Docker.

Ένα στιγμιότυπο περιέκτη είναι ουσιαστικά μια στατική αναπαράσταση που καθορίζει την εκτέλεση ενός περιέκτη. Αυτό σημαίνει ότι περιέχει πληροφορίες σχετικά τόσο με την δομή του συστήματος αρχείων υπό τη μορφή περιέκτη (containerized filesystem) όσο και με το ποιες διεργασίες θα τρέξουν εντός του περιέκτη. Με λίγα λόγια, ένα στιγμιότυπο του περιέκτη είναι ένα αμετάβλητο αρχείο που ουσιαστικά περιγράφει μια στιγμιαία κατάσταση του περιέκτη.

Το σύστημα αρχείων του στιγμιότυπου δημιουργείται στοιβάζοντας μια λίστα επιπέδων μόνο-για-ανάγνωση, χρησιμοποιώντας ένα ενωτικό σύστημα αρχείων. Έπειτα, όταν ένας περιέκτης αρχικοποιείται από αυτό το στιγμιότυπο, ένα λεπτό εγγράψιμο επίπεδο προστίθεται πάνω από τα μόνο-για-ανάγνωση επίπεδα. Αυτό το επίπεδο ονομάζεται επίσης το «επίπεδο του περιέκτη». Όλες οι αλλαγές που γίνονται στον περιέκτη που εκτελείται, όπως η εγγραφή νέων αρχείων, η τροποποίηση υπάρχοντων αρχείων και η διαγραφή αρχείων, εγγράφονται σε αυτό το λεπτό εγγράψιμο επίπεδο του περιέκτη.

### 2.2.4.2 Μητρώα

Εφόσον τα στιγμιότυπα είναι ουσιαστικά αρχεία προδιαγραφών περιεκτών, μπορούν να αποκτήσουν έκδοση (versioned), να μεταφορτωθούν και να μοιραστούν σε χρήστες. Αυτά τα κεντρικά σημεία όπου τα στιγμιότυπα θα μεταφορτώνονταν και θα φιλοξενούνταν ονομάζονται μητρώα. Συγκεκριμένα το Docker έχει εφαρμόσει το δικό του μητρώο: DockerHub [17]. Τα μητρώα λειτουργούν σαν ιδέα πολύ παρόμοια με τα αποθετήρια του GitHub, με μόνη εξαίρεση ότι λειτουργούν συγκεκριμένα για στιγμιότυπα, όχι για οποιονδήποτε τύπο κώδικα. Οι χρήστες μπορούν να ανεβάσουν τα στιγμιότυπα τους, να τους δώσουν έκδοση (version), μέχρι και να έχουν διαφορετικές διακλαδώσεις, όπως ακριβώς και στο GitHub.



## 2.3 Κυβερνήτης

Οι περιέκτες παρέχουν ένα τρόπο για εφαρμογές να εκτελούνται εντός απομονωμένων, αμετάβλητων και αναπαράξιμων περιβαλλόντων. Η εκκίνηση ενός περιέκτη είναι τριτογενής, ό,τι πρακτικά κάνει κάθε developer σε τακτική βάση. Το υλικοτεχνικό πρόβλημα παρουσιάζεται όταν ο αριθμός των εφαρμογών (και χρηστών) αυξάνει σημαντικά. Σε αυτή την περίπτωση, η διαχείριση ενός σημαντικού αριθμού φυσικών κόμβων που εκτελούν περιέκτες χρηστών, η διενέργεια ελέγχων υγείας σε αυτά και η εξασφάλιση επαναφοράς περιεκτών από αποτυχία δεν είναι επουσιώδης εργασία.

Ο **Κυβερνήτης** [7] ικανοποιεί αυτή την ανάγκη, προσφέροντας επιπλέον τρόπους δυναμικής κλιμάκωσης εφαρμογών και τρόπους ώστε διαφορετικοί περιέκτες να επικοινωνούν μεταξύ τους και να μοιράζονται υποκείμενο αποθηκευτικό χώρο. Είναι μια πλατφόρμα διαχείρισης φόρτων εργασιών υπό τη μορφή περιεκτών, και είναι ευρέως διαδεδομένος στο σημερινό τοπίο του υπολογιστικού νέφους.

### 2.3.1 Επισκόπηση

Η κύρια φιλοσοφία πίσω από τον Κυβερνήτη είναι ότι μπορεί κανείς να ορίσει δηλωτικά την επιθυμητή κατάσταση του συστήματος, και το σύστημα θα αυτοεπιβλέπεται διαρκώς και θα προσπαθεί να επιτύχει αυτήν την κατάσταση. Η κατάσταση εκφράζεται σαν ένα σετ αντικειμένων *YAML* [18] τα οποία διατηρούνται σε μία διανεμημένη, υψηλής διαθεσιμότητας βάση κλειδιών-τιμών, που ονομάζεται *etcd* [19].

### 2.3.2 Ελεγκτές και Αντικείμενα

Ο Κυβερνήτης περιλαμβάνει έναν αριθμό αφηρημένων εννοιών που αναπαριστούν την κατάσταση του συστήματος. Αυτές οι αφηρημένες έννοιες αναπαριστώνται από αντικείμενα στο API του Κυβερνήτη. Ένα αντικείμενο του Κυβερνήτη είναι μία «καταγραφή πρόθεσης». Μόλις ο χρήστης δημιουργήσει ένα αντικείμενο, το σύστημα του Κυβερνήτη θα δουλέψει διαρκώς για να εξασφαλίσει ότι το αντικείμενο υπάρχει και έχει την επιθυμητή κατάσταση.

Κάθε αντικείμενο στον Κυβερνήτη θα έχει κάποιο από τα ακόλουθα πεδία:

- **Kind:** Το είδος του αντικειμένου. Τα αντικείμενα μπορεί να είναι, για παράδειγμα, τύπου: Pod, Deployment, Service και άλλα.
- **apiVersion:** Προσδιορίζει την έκδοση του αντικειμένου.
- **Metadata:** Δεδομένα που βοηθούν στην μοναδική ταυτοποίηση του αντικειμένου, συμπεριλαμβανομένων ενός αλφαριθμητικού ονόματος, μιας UID και ένας προαιρετικός χώρος ονόματος.
- **Spec και Status:** Κάθε αντικείμενο του Κυβερνήτη περιλαμβάνει δύο ένθετα πεδία αντικειμένων που διέπουν τη διαμόρφωση του αντικειμένου: το *spec* και το *status*. **Το spec, το οποίο παρέχει ο χρήστης, περιγράφει την επιθυμητή κατάσταση και το status περιγράφει την πραγματική κατάσταση του αντικειμένου.** Σε οποιαδήποτε στιγμή, το Control Plane του Κυβερνήτη διαχειρίζεται ενεργά την πραγματική κατάσταση ενός αντικειμένου ώστε να αντιστοιχεί με την επιθυμητή κατάσταση που ο χρήστης.

### 2.3.3 Αποθηκευτικός Χώρος

#### 2.3.3.1 Λογικοί Δίσκοι

Ένα *Pod* που χρησιμοποιεί λογικούς δίσκους προσδιορίζει στο πεδίο του *spec* ποιους λογικούς δίσκους σκοπεύει να χρησιμοποιήσει, όπως και το μονοπάτι στο οποίο αυτοί οι δίσκοι θα προσαρτηθούν στα συστήματα αρχείων του περιέκτη. Διεργασίες που εκτελούνται εντός ενός περιέκτη «βλέπουν» ένα σύστημα αρχείων αποτελούμενο από το στιγμιότυπο τους του Docker και τους προσαρτημένους δίσκους τους. Το στιγμιότυπο Docker είναι στη ρίζα της ιεραρχίας του συστήματος αρχείων, και όποιοι δίσκοι προσαρτώνται στα προσδιορισμένα μονοπάτια εντός του στιγμιότυπου.

#### 2.3.3.2 *PersistentVolumes* και *PersistentVolumeClaims*

Ο Κυβερνήτης παρέχει στους χρήστες το υποσύστημα *PersistentVolume* μέσω του API του ώστε να αφαιρέσει τις λεπτομέρειες του πώς παρέχεται ο αποθηκευτικός χώρος από το πώς καταναλώνεται. Για αυτό παρέχει τα Αντικείμενα API *PersistentVolume* και *PersistentVolumeClaim*.

Ένα *PersistentVolume* είναι μία οντότητα που αναπαριστά κάποιο κομμάτι αποθηκευτικού χώρου στη συστάδα που έχει προβλεφθεί, είτε στατικά από ένα χρήστη είτε δυναμικά. Είναι ένας πόρος στη συστάδα όπως ένας κόμβος είναι ένας πόρος της συστάδας.

Ένα *PersistentVolumeClaim* είναι ένα αίτημα ενός χρήστη να καταναλώσει αποθηκευτικό χώρο. Ακριβώς όπως ένα *Pod* ζητά να καταναλώσει έναν πόρο ενός Κόμβου, ένα *PersistentVolumeClaim* ζητά να καταναλώσει έναν πόρο ενός *PersistentVolume*.

## 2.4 Kubeflow

### 2.4.1 Επισκόπηση

Το Kubeflow [3] είναι μία εργαλειοθήκη μηχανικής μάθησης για τον Κυβερνήτη που αποσκοπεί στην απλοποίηση της κλιμάκωσης και της διανομής μοντέλων μηχανικής μάθησης στην παραγωγή, αξιοποιώντας τα πλεονεκτήματα του Κυβερνήτη.

Το Kubeflow ξεκίνησε σαν μεταβολή σε ανοιχτό κώδικα του τρόπου που η Google έτρεχε το *TensorFlow* [20] εσωτερικά, βασισμένο σε μία διοχέτευση που ονομάζεται *TensorFlow Extended* [21]. Ξεκίνησε σαν ένας απλούστερος τρόπος ώστε να εκτελούνται εργασίες *TensorFlow* στον Κυβερνήτη, αλλά έχει έκτοτε επεκταθεί σε ένα πολυ-αρχιτεκτονικό, πολυ-νεφικό πλαίσιο για την εκτέλεση ροών εργασιών μηχανικής μάθησης από άκρη σε άκρη.

Το Kubeflow είναι ένα σετ από *CustomResourceDefinitions* και εφαρμογές ιστού για το χειρισμό αυτόν, καθώς επίσης και το Central Dashboard, το οποίο τα συνδέει όλα μαζί για να προσφέρει μία συνεκτική εμπειρία. Αυτό περιλαμβάνει τα τμήματα Jupyter Notebooks και Pipelines που θα εκθέσουμε στην επόμενη υποενότητα. Ο χρήστης αναμένεται να αλληλεπιδράσει με το Kubeflow μέσω αυτών των UI.

Στις ακόλουθες υποενότητες, θα παρουσιάσουμε τα κεντρικά τμήματα του Kubeflow.

### 2.4.2 Jupyter Notebooks

Αυτό το τμήμα είναι υπεύθυνο για να επιτρέπει στο χρήστη να διανέμει και να χειρίζεται Jupyter Notebooks στην Kubeflow συστάδα τους. Για να το επιτύχει, προσφέρει ένα φιλικό προς το χρήστη UI που επιτρέπει στο χρήστη να διαχειρίζεται τον κύκλο ζωής των *CustomResources* του Notebook.

### 2.4.3 Διοχετεύσεις (Pipelines)

Το Kubeflow Pipelines (KFP) [10] είναι μια πλατφόρμα για το χτίσιμο και την διανομή φορητών και κλιμακώσιμων φόρτων εργασιών μηχανικής μάθησης βασισμένα σε περιέκτες Docker και είναι ένα από τα κεντρικά τμήματα του Kubeflow. Διανέμεται αυτόματα κατά τη διανομή του Kubeflow. Η πλατφόρμα Kubeflow Pipelines αποτελείται από μία διεπαφή χρήστη (user interface, UI) για τη διαχείριση και την παρακολούθηση πειραμάτων, εργασιών και εκτελέσεων, μαζί με μία μηχανή για τον προγραμματισμό ροών εργασιών μηχανικής μάθησης πολλαπλών βημάτων. Συνοδεύεται επίσης από ένα SDK για τον ορισμό και το χειρισμό διοχετεύσεων και τμημάτων. Εκτός αυτού, υπάρχουν notebooks (τετράδια) για την αλληλεπίδραση με το σύστημα χρησιμοποιώντας το SDK.

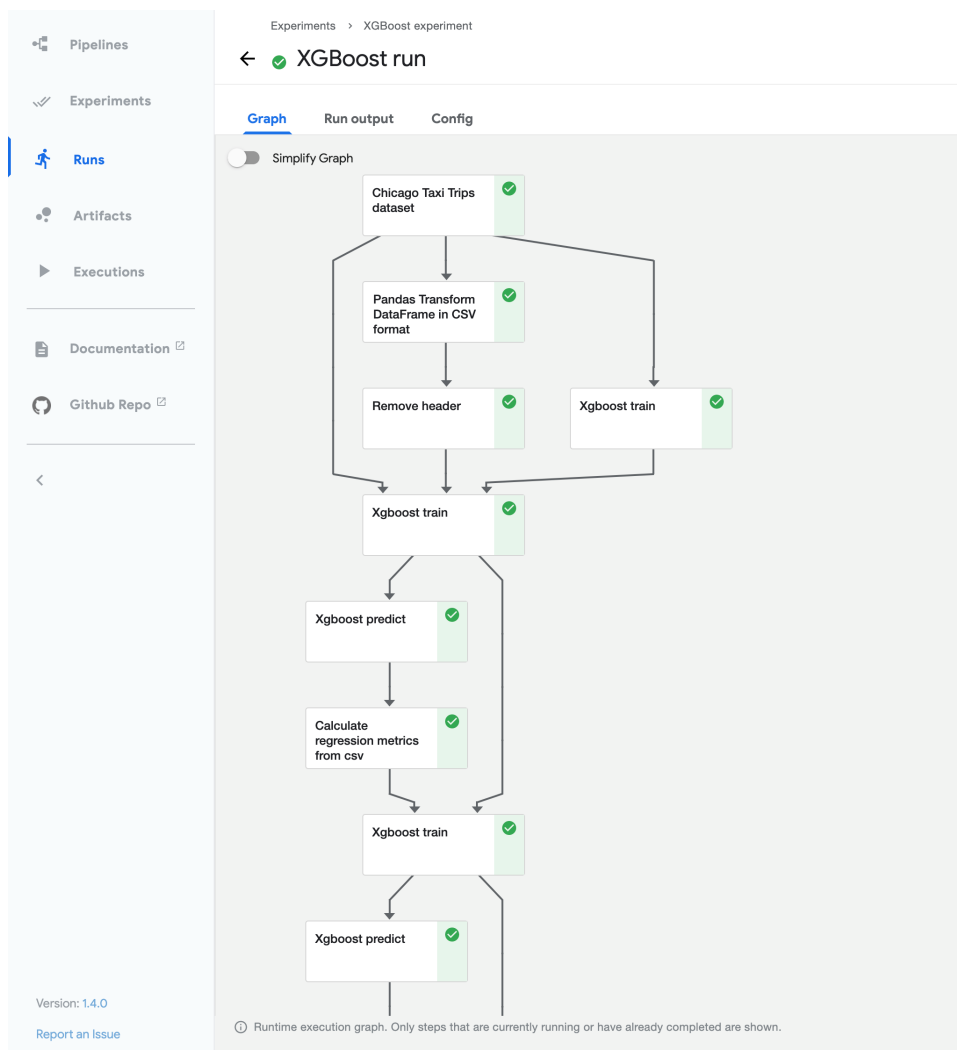


Figure 2.1: Το γράφημα χρόνου εκτέλεσης μιας διοχέτευσης στο KFP UI

Το KFP προσφέρει εννοχρήστρωση από άκρη σε άκρη, επιτρέποντας και απλουστεύοντας την εννοχρήστρωση διοχετεύσεων μηχανικής μάθησης. Επιπλέον, είναι εύκολο για τους χρήστες να δοκιμάσουν πολυάριθμες ιδέες και να διαχειριστούν διάφορες δοκιμές/πειράματα. Επιπροσθέτως, επιτρέπει την επαναχρησιμοποίηση τμημάτων και διοχετεύσεων για την γρήγορη δημιουργία λύσεων από άκρη σε άκρη χωρίς την ανάγκη χτισίματος κάθε φορά.

### 2.4.3.1 Argo

Το Argo [22] είναι μία μηχανή ροών εργασιών. Είναι μία επέκταση της συστάδας του Κυβερνήτη που καθιστά δυνατή την εκτέλεση ροών εργασιών. Ο χρήστης υποβάλλει έναν ορισμό *YAML* μιας ροής εργασιών (*Workflow CustomResourceDefinition*) και στη συνέχεια το Argo είναι υπεύθυνο ώστε να εκκινεί εργασίες με την κατάλληλη σειρά και να αναμένει μέχρι την περάτωσή τους. Προσφέρει επίσης ένα εργαλείο *Command Line Interface (CLI)*, καθώς επίσης και ένα βασικό *User Interface (UI)* για την εικονικοποιημένη απεικόνιση των ροών εργασιών.

**Τα Kubeflow Pipelines χρησιμοποιούν το Argo σαν τη μηχανή ροών εργασιών τους.** Το Software Development Kit (SDK) μεταγλωττίζει τον πηγαίο κώδικα του χρήστη σε

ένα *Argo Workflow CustomResourceDefinition* που πρέπει μετά να εφαρμοστεί στη συστάδα.

### 2.4.4 Katib

Το Katib [4] είναι το τμήμα του Kubeflow για την βελτιστοποίηση των υπερπαραμέτρων των μοντέλων. Η κύρια ιδέα πίσω από το Katib είναι ότι ορίζει τη διαδικασία εκπαίδευσης εντός ενός περιέκτη και εκτελεί αυτή τη λογική πολλαπλές φορές ώστε να καταλήξει σε ένα ικανοποιητικό σετ υπερπαραμέτρων. Αυτό επιτυγχάνεται με τη δημιουργία ενός *Experiments CustomResourceDefinition* και την απαίτηση ο κώδικας να γίνει υπό τη μορφή περιέκτη με συγκεκριμένο τρόπο.

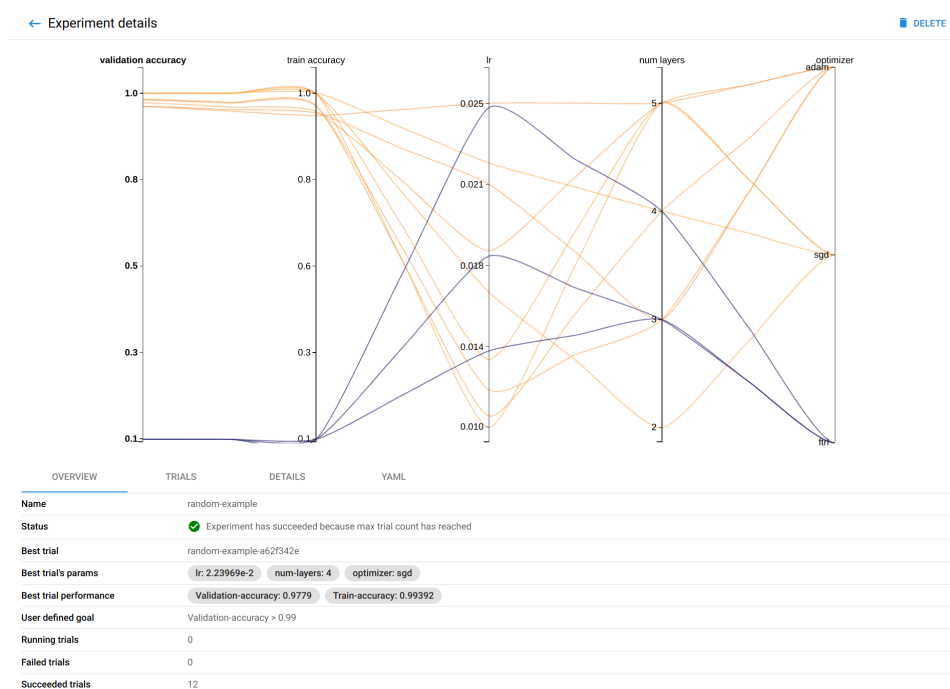


Figure 2.2: Παράδειγμα γραφήματος του Katib UI που δείχνει το επίπεδο επαλήθευσης και εκπαιδευτικής ακριβείας για διάφορους συνδυασμούς τιμών υπερπαραμέτρων

Ο κώδικας υπό τη μορφή περιέκτη θα πρέπει να μπορεί να τρέξει αυτόνομα. Αυτό σημαίνει ότι ο περιέκτης που δημιουργήθηκε θα πρέπει να μπορεί τόσο να προσπελάσει τα δεδομένα όσο και να εκπαιδεύσει το μοντέλο. Η είσοδος στον κώδικα θα είναι οι τιμές των υπερπαραμέτρων, είτε στο πεδίο *args*, είτε σαν μεταβλητές περιβάλλοντος, και θα εξάγει σαν αποτέλεσμα μία μετρική.

Οι περιέκτες που δομούνται με αυτόν τον τρόπο μπορούν να τρέξουν πολλαπλές φορές, ακόμη και παράλληλα, με διαφορετικές τιμές υπερπαραμέτρων σαν εισόδους, **σε αναζήτηση ενός σετ αυτών που θα επιτύχουν έναν προσδιορισμένο στόχο επίδοσης**. Υπάρχουν πολλαπλές στρατηγικές αναζήτησης μέσα στο χώρο υπερπαραμέτρων, όπως: Διασταυρωμένη Επαλήθευση (Cross-Validation), Τυχαία αναζήτηση, Μπεϋζιανή βελτιστοποίηση για να ονομάσουμε μερικές.

Τα αποτελέσματα κάθε εκτέλεσης αποθηκεύονται επίσης σε μία κεντρική βάση δεδομένων που παραμένει με τη χρήση της *PersistentVolumes*. Όλα αυτά ενορχηστρώνονται από το *Experiment CustomResource Controller* που είναι υπεύθυνο για τη διανομή Jobs Κυβερνήτη,

την εκπαίδευση του μοντέλου, την καταγραφή της επίδοσης κάθε εκτέλεσης, την εφαρμογή του αλγορίθμου αναζήτησης και τέλος την απόφαση παύσης της διεργασίας βελτιστοποίησης.

### 2.4.5 MiniKF

Το MiniKF [23] είναι ένα αντικείμενο μονού κόμβου του KubeFlow, που μπορεί να διανεμηθεί τοπικά ή στο νέφος. Συνδυάζει το KubeFlow με την πλατφόρμα Rok Data Management, την οποία θα περιγράψουμε στην επόμενη υποενότητα.

#### 2.4.5.1 Rok

Το Rok [24] είναι μια πλατφόρμα διαχείρισης και αποθήκευσης δεδομένων που επιτρέπει στους χρήστες λάβουν στιγμιότυπα, να δώσουν έκδοση, να πακετάρουν, να διανείμουν και να κλωνοποιήσουν το πλήρες περιβάλλον τους μαζί με τα δεδομένα του. Είναι εγγενώς ενσωματωμένο στον Κυβερνήτη ως μία από τις υποστηριζόμενες πλατφόρμες του.

Είναι σημαντικό να σημειώσουμε ότι στην δουλειά μας για μία κατανομημένη διεργασία AutoML στον Κυβερνήτη, αξιοποιούμε τη λειτουργικότητα του Rok να λαμβάνει στιγμιότυπα του λογικού δίσκου σε κάθε βήμα της διεργασίας, καθιστώντας τα ενδιάμεσα και τελικά αποτελέσματα των πειραμάτων μας πλήρως αναπαράξιμα.

## 2.5 Kale και Kale-SDK

### 2.5.1 Επισκόπηση

Το Kale ([11]) σημαίνει "KubeFlow Automated pipeLines Engine" και είναι ένα πρότζεκτ που στοχεύει στην απλοποίηση της εμπειρίας Επιστήμης Δεδομένων στη διανομή ροών εργασιών KubeFlow Pipelines. Επεκτείνοντας το Jupyter UI, επιτρέπει στους χρήστες να διανέμουν Jupyter Notebooks, τα οποία εκτελούνται τοπικά ή στο νέφος, σε KubeFlow Pipelines. Αυτό μπορεί να συμβεί επισημαίνοντας κελιά κώδικα και κάνοντας κλικ σε ένα κουμπί διανομής στο επεκτεταμένο Jupyter UI. Το Kale είναι υπεύθυνο για τη μετατροπή του επισημασμένου Notebook του χρήστη σε ένα λειτουργικό KubeFlow Pipeline, όπως και για να φροντίσει για τη διαβίβαση δεδομένων μεταξύ των βημάτων και τη διαχείριση του κύκλου ζωής του KubeFlow Pipeline.



Figure 2.3: Το Λογότυπο του Kale

Εκτός από την επέκταση του Jupyter UI που περιγράψαμε παραπάνω, το Kale παρέχει ένα Software Development Kit, στο οποίο θα αναφερόμαστε ως το **Kale SDK** στο εξής. Το Kale SDK επιτρέπει στους χρήστες να γράφουν κώδικα Python με βάση συναρτήσεις και

να τον μετατρέπουν σε πλήρως αναπαράξιμα KubeFlow Pipelines χωρίς να πραγματοποιούν οποιαδήποτε αλλαγή στον αρχικό πηγαίο κώδικα. Για τους σκοπούς αυτής της διπλωματικής εργασίας, επεκτείναμε το Kale SDK ώστε να μπορεί να δημιουργεί πειράματα AutoML στο KubeFlow. Ας δώσουμε τώρα επιγνώση κάποιων βασικών ιδεών του Kale-SDK που είναι απαραίτητες για την κατανόηση της δουλειάς μας.

Για μία σύντομη και ενδελεχή επισκόπηση της ιστορίας του Kale και των βασικών λειτουργιών ανατρέξτε στο εξαιρετικό blog post από τον *αρχικό συγγραφέα του Kale, Stefano Fioravanzo* [25]. Μπορείτε επίσης να ανατρέξετε στην επίσημη τεκμηρίωση του Kale-SDK [26]

### 2.5.2 Η Κλάση Step

Η κλάση *Step* βρίσκεται στο δομοστοιχείο *step* του Kale, και επιτρέπει στους χρήστες να δηλώνουν συναρτήσεις Python σαν βήματα του Kale. Ένα βήμα του Kale είναι ουσιαστικά ένα καλούμενο αντικείμενο που περιτυλίγει μία ορισμένη από το χρήστη συνάρτηση με την αποθηκευτική λογική του Kale (2.5.7).

Οι χρήστες μπορούν να αρχικοποιήσουν ένα αντικείμενο Step χρησιμοποιώντας τον διακοσμητή *@step*, που μπορεί να εισαχθεί από το δομοστοιχείο *sdk* του Kale. Όταν περιτυλίγεται γύρω από μία συνάρτηση Python, ο διακοσμητής *@step* επιστρέφει ένα αντικείμενο Step του οποίου η *do\_run* ιδιότητα έχει παρακαμφθεί από την ορισμένη από το χρήστη συνάρτηση Python. Το ακόλουθο απόσπασμα κώδικα δίνει ένα τέτοιο παράδειγμα.

Listing 2.1: Παράδειγμα συνάρτησης βήματος διοχέτευσης, διακοσμημένη με τον διακοσμητή *@step*

```
1 @step(name="my_step")
2 def step_1(in_1, in_2):
3     # implement the step's business logic here
```

Ένας άλλος τρόπος να οριστεί ένα βήμα του Kale είναι μέσω της χρήσης της κλάσης *Step* σε υποκλάση και της εφαρμογής μιας μεθόδου *do\_run*. Ορίστε ένα παράδειγμα κώδικα.

Listing 2.2: Παράδειγμα βήματος του Kale που χρησιμοποιεί την κλάση *Step* σαν υποκλάση

```
1 class ExampleStep(Step):
2     name = "example-step"
3     def do_run(self, param1, param2):
4         # implement the step's business logic here
```

Για τους σκοπούς αυτής της εργασίας, ορίσαμε έναν αριθμό κατά παραγγελία βημάτων χρησιμοποιώντας την τεχνική χρήσης υποκλάσεων που περιγράψαμε παραπάνω.

Οι παράμετροι εισόδου ανιχνεύονται αυτόματα αναλύοντας τη μέθοδο *do\_run* του βήματος.

### 2.5.3 Η Κλάση Pipeline

Η κλάση *Pipeline* βρίσκεται στο δομοστοιχείο *pipeline* του Kale και χρησιμοποιείται για να ορίσει ένα Kale Pipeline, τα βήματά του και όλες τις εξαρτήσεις του. Επεκτείνει την

κλάση *DiGraph* του *networkx* ([27]) για κατευθυνόμενα γραφήματα με βρόχους ώστε να εκμεταλλευτεί τους υποκείμενους σχετικούς με γραφήματα αλγορίθμους, αλλά παρέχει επίσης βοηθητικές συναρτήσεις ώστε να λειτουργεί αντικείμενα *Step* του Kale αντί των βασικών *networkx* «κόμβων». Αυτό καθιστά απλούστερη την πρόσβαση στα βήματα της διοχέτευσης και στις ιδιότητές τους.

Ένα αντικείμενο Pipeline του Kale μπορεί να μετατραπεί σε ένα KubeFlow Pipeline χρησιμοποιώντας την κλάση *Compiler* (2.5.6) του Kale.

Οι χρήστες του Kale-SDK δεν αναμένονται να χρησιμοποιούν τα αντικείμενα *Pipeline* άμεσα. Αντ' αυτού, το δομοστοιχείο *api* του Kale-SDK παρέχει έναν διακοσμητή *@pipeline* που επιτρέπει στους χρήστες να δηλώσουν εύκολα ένα Kale Pipeline «περιτυλίγοντας» τον διακοσμητή γύρω από τη συνάρτηση διοχέτευσης.

Listing 2.3: Παράδειγμα συνάρτησης διοχέτευσης διακοσμημένη με τον διακοσμητή *@pipeline*

```
1 @pipeline(name="my_pipeline", experiment="test")
2 def pipeline(param1="dont"):
3     res1 = step_1(param)
4     if param == "do":
5         res2 = step_2(res1)
```

#### 2.5.4 Η Γλώσσα Συγκεκριμένου Τομέα του Kale

Η γλώσσα συγκεκριμένου τομέα (domain specific language) του Kale-SDK αποσκοπεί στο να προσφέρει ένα API με τη μορφή της Python για τον ορισμό μιας διοχέτευσης. Από εδώ και στο εξής θα χρησιμοποιούμε το ακρωνύμιο: «DSL» για να αναφερθούμε στον όρο: «γλώσσα συγκεκριμένου τομέα» (domain specific language).

Η ουσία της DSL του Kale είναι ότι επιτρέπει τη συγγραφή συναρτήσεων Python που περιγράφουν την αρχιτεκτονική μιας διοχέτευσης και οι οποίες μπορούν, χωρίς διακοσμητή *@pipeline*, να τρέξουν ως είναι, τοπικά. Η μόνη απαιτούμενη διαδικασία για τη μετατροπή της σε αντικείμενα *Pipeline* θα πρέπει να είναι η εφαρμογή του διακοσμητή *@pipeline*.

Η DSL προς το παρόν επιτρέπει τις ακόλουθες εντολές Python:

1. Κλήσεις συναρτήσεων με παραμέτρους εισόδου που είναι παράμετροι διοχέτευσης ή άλλοι έξοδοι κλήσεων συναρτήσεων. Αυτές οι κλήσεις συναρτήσεων αντιστοιχούν σε βήματα της διοχέτευσης.

Listing 2.4: Παράδειγμα βήματος με παράμετρο εισόδου που είναι παράμετρος διοχέτευσης

```
1 def dsl(param="Hello"):
2     step_1(param)
3     step_2()
```

2. Αναθέσεις από κλήσεις συναρτήσεων (βήματος). Οι ανατεθείσες τιμές είναι οι έξοδοι βήματος. Πολλαπλές έξοδοι μπορούν να ανακτηθούν με αναθέσεις πλειάδων.

Listing 2.5: Παράδειγμα βήματος όπου η παράμετρος εισόδου είναι η έξοδος ενός άλλου βήματος



```

1  def dsl(param="Hello"):
2      my_out = step_1(param)
3      step_2(my_out)

```

3. Εντολές `if` με boolean συνθήκες. Αυτές οι συνθήκες πρέπει να έχουν μόνο σύγκριση μεταξύ παραμέτρων διοχέτευσης και σταθερών τιμών.

Listing 2.6: Παράδειγμα βημάτων εντός εντολών `if`

```

1  def dsl(param1="no", param2="no"):
2      res1 = step_1(param)
3      if param == "yes":
4          res2 = step_2(res1)
5      if param2 == "yes":
6          step_3(res2)

```

### 2.5.5 Η Κλάση *PythonProcessor*

Η κλάση *PythonProcessor* είναι μία κλάση του API που βρίσκεται στο δομοστοιχείο *processors* του Kale και αποσκοπεί στην επαλήθευση μιας συνάρτησης Python γραμμένης στην DSL του Kale και τη μετατροπή της σε αντικείμενο Pipeline. Ο κατασκευαστής του *PythonProcessor* χρειάζεται κυρίως δύο ορίσματα εισόδου:

- **pipeline\_function (Callable):** Μία συνάρτηση διοχέτευσης, γραμμένη στην DSL του Kale (2.5.4). Αυτή η συνάρτηση ουσιαστικά περιγράφει ολόκληρη την αρχιτεκτονική της διοχέτευσης, δηλαδή τη ροή δεδομένων από το πρώτο ως το τελευταίο βήμα της διοχέτευσης.
- **config (pipeline.PipelineConfig):** Αυτό είναι ένα αντικείμενο διαμόρφωσης, που βρίσκεται στο δομοστοιχείο *pipeline* του Kale που χρησιμοποιείται για την αποθήκευση μεταδεδομένων διοχέτευσης, όπως το όνομα της διοχέτευσης, το όνομα του πειράματος διοχέτευσης του KubeFlow, μια περιγραφή της διοχέτευσης και άλλα τέτοια μεταδεδομένα.

Η επαλήθευση της συνάρτησης εισόδου διοχέτευσης συμβαίνει **κατά την αρχικοποίηση του αντικειμένου *PythonProcessor***, ενώ η μέθοδος ***run*** του αντικειμένου είναι υπεύθυνη για τη μετατροπή της συνάρτησης εισόδου διοχέτευσης σε ένα αντικείμενο *Pipeline* (2.5.3).

### 2.5.6 Η Κλάση *Compiler*

Η εσωτερική κλάση *Compiler*, μετατρέπει ένα αντικείμενο *Pipeline* του Kale σε ένα KFP Pipeline. Όταν χρησιμοποιείται το Kale για την εκτέλεση μίας διακοσμημένης συνάρτησης `@pipeline`, το Kale πρώτα δημιουργεί ένα αντικείμενο Pipeline, μέσω του *PythonProcessor* που περιγράψαμε στην υποενότητα 2.5.5, και μετά χρησιμοποιεί ένα αντικείμενο *Compiler* για να το μετατρέψει σε ένα KFP Pipeline.

Η διεργασία μετατροπής ενός αντικειμένου Pipeline του Kale σε ένα KubeFlow Pipeline υλοποιείται από τη μέθοδο `compile_and_run` του *Compiler*, η οποία:

1. Μεταγλωττίζει το αντικείμενο *Pipeline* σε ένα *Workflow YAML*.
2. Δημιουργεί ένα KubeFlow Pipeline μεταφορώνοντας το *Workflow* στο KFP.

### 2.5.7 Μηχανισμός Διαβίβασης Δεδομένων του Kale

Ούτως ώστε να διαβιβάσει δεδομένα μεταξύ βημάτων, το Kale προμηθεύει αυτόματα ένα νέο *PersistentVolume* ή χρησιμοποιεί έναν υπάρχοντα λογικό δίσκο του χώρου εργασίας που επισυνάπτεται σε κάθε περιέκτη του βήματος μίας διοχέτευσης.

Το Kale προσθέτει κώδικα στο τέλος της εκτέλεσης της συνάρτησης *do\_run* ενός βήματος ώστε να αποθηκεύσει τα αντικείμενα εξόδου του βήματος σε αυτό το κοινό *PersistentVolumeClaim* κατά την εκτέλεση. Παρόμοια, προσθέτει κώδικα στην αρχή της εκτέλεσης ενός βήματος, ώστε να φορτώσει τα αντικείμενα εξόδου των προηγούμενων βημάτων και να τα δώσει ως εισόδους στο τρέχων βήμα.

Για αυτό το σκοπό, το Kale χρησιμοποιεί το δομοστοιχείο του *marshal*, και πιο συγκεκριμένα τις μεθόδους **save** και **load** του δομοστοιχείου που χρησιμοποιούνται για να αποθηκεύσει και να φορτώσει τα δεδομένα αντίστοιχα.

### 2.5.8 Εκδόσεις και Στιγμιότυπα Δεδομένων στο Kale

Στην περίπτωση που το Kale εκτελείται σε έναν Notebook Server εντός ενός αντικειμένου MiniKF (2.4.5), χρησιμοποιεί τον πελάτη Rok (2.4.5.1) για:

1. Να αναγνωρίσει υπάρχοντες λογικούς δίσκους χώρου εργασίας/δεδομένων στον Notebook Server, να λάβει στιγμιότυπά τους και να τους προσαρτήσει στα βήματα της διοχέτευσης. Με αυτόν τον τρόπο, ο χώρος εργασίας του χρήστη (που ενδεχομένως περιέχει αρχεία δεδομένων ή εγκατεστημένες εξαρτήσεις) διατηρείται στην τρέχουσα διοχέτευση.
2. Να λάβει στιγμιότυπα λογικών δίσκων στο τέλος της εκτέλεσης της διοχέτευσης, προσφέροντας έναν πρακτικό τρόπο ανάκτησης αποθηκευμένων αντικειμένων που παράχθηκαν κατά την εκτέλεση της διοχέτευσης, **ως αντικείμενα πλήρως εκπαιδευμένων μοντέλων ή επεξεργασμένα δεδομένα**.
3. Να λάβει στιγμιότυπα λογικών δίσκων στην αρχή εκτέλεσης κάθε βήματος, προσφέροντας έναν πρακτικό τρόπο ανάκτησης της κατάστασης των δεδομένων πριν από μία ενδεχόμενη αποτυχία βήματος.

## 2.6 Machine Learning Meta-Data

Η Machine Learning Meta-Data (MLMD) [28] είναι μία βάση δεδομένων NoSQL όπου καταγράφουμε και ανακτούμε μεταδεδομένα σχετιζόμενα με ροές εργασιών και πειράματα μηχανικής μάθησης. Από εδώ και στο εξής, θα αναφερόμαστε στη βάση Machine Learning Meta-Data ως "MLMD".

Το MLMD ορίζει οντότητες που μπορούν να αποθηκευθούν στην ή να ανακτηθούν από τη βάση δεδομένων. Στη δουλειά μας, σχετίζουμε κάθε μία από αυτές τις οντότητες με μία συγκεκριμένη ιδέα μιας ροής εργασιών ή ενός πειράματος μηχανικής μάθησης. Οι κύριες οντότητες που χρησιμοποιούμε στη δουλειά μας είναι:

- **Contexts:** Ένα *Context* είναι μια οντότητα που αποσκοπεί στο να εμπεριέχει μια ομάδα άλλων οντοτήτων που όλες μαζί ορίζουν ένα γενικότερο πλαίσιο. Αντιστοιχούμε αυτές τις οντότητες σε KFP Runs.
- **Executions:** Ένα *Execution* αντιστοιχεί σε κάτι που εκτελείται. Ένα *Execution* μπορεί να μέρος ενός (ή περισσότερων) *Contexts*. Αντιστοιχούμε αυτές τις οντότητες σε βήματα KFP.
- **Artifacts:** Ένα *Artifact* είναι κάτι που καταναλώνεται ή παράγεται από ένα *Execution*. Επιπλέον, μπορεί να μέρος ενός (ή περισσότερων) *Contexts*. Στη δουλειά μας παράγουμε *Artifacts* που αντιστοιχούν σε έναν αριθμό οντοτήτων σχετιζόμενων με τη μηχανική μάθηση.
- **Attributions:** Ένα *Attribution* δηλώνει ότι ένα *Artifact* είναι μέρος μιας οντότητας *Context*.
- **Associations:** Ένα *Association* δηλώνει ότι ένα *Execution* είναι μέρος μιας οντότητας *Context*.
- **Events:** Ένα *Event* δηλώνει ότι ένα *Artifact* είναι είσοδος ή έξοδος μιας οντότητας *Execution*.

Ανατρέξτε στους οδηγούς χρηστών του MLMD του TensorFlow [29] για μια λεπτομερή επεξήγηση των ιδεών πίσω από αυτές τις οντότητες.

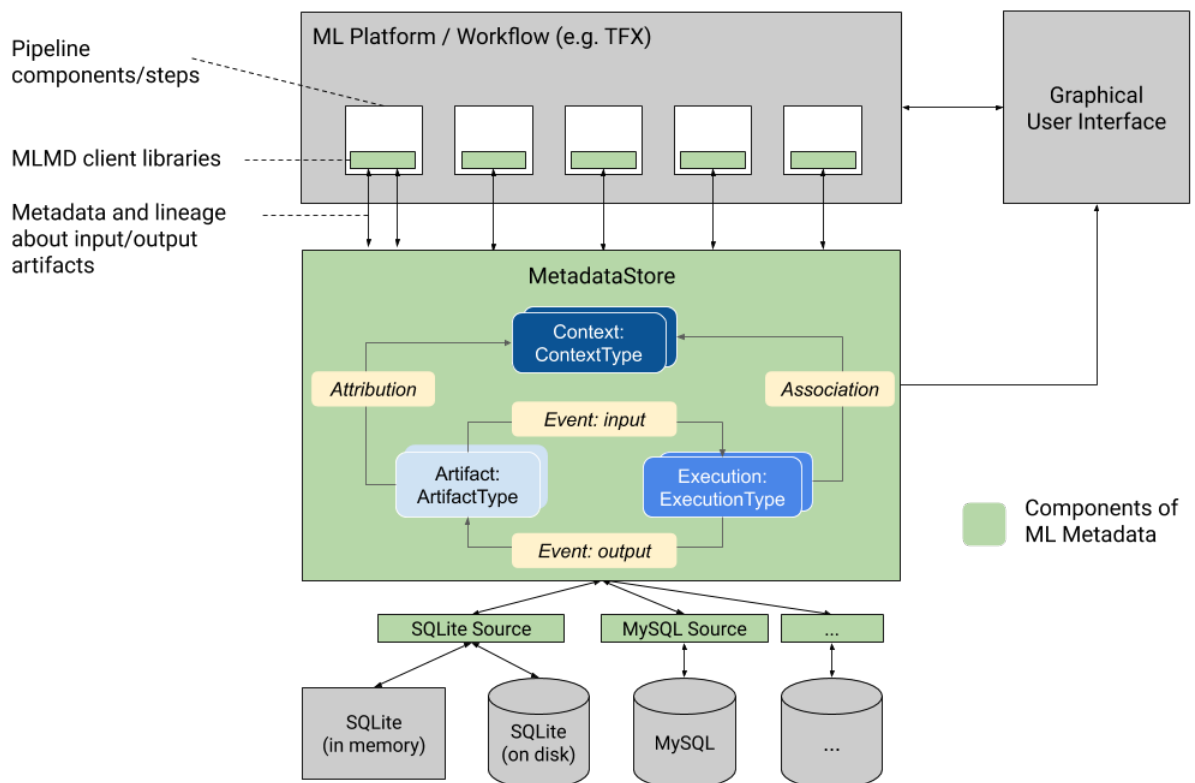


Figure 2.4: Μια Επισκόπηση του MLMD



# Η Μελέτη μας πάνω στον Μηχανισμό Μετα-Μάθησης του *Auto-sklearn*

---

## 3.1 Επισκόπηση

Το πακέτο *auto-sklearn* ([1]) είναι μια αυτοματοποιημένη εργαλειοθήκη μηχανικής μάθησης που απαλλάσσει ένα χρήστη μηχανικής μάθησης από την επιλογή αλγορίθμου και τη ρύθμιση υπερπαραμέτρων. Αξιοποιώντας πρόσφατα πλεονεκτήματα στη Μπεϋζιανή βελτιστοποίηση, τη μεταμάθηση και την κατασκευή ενωμένων μοντέλων, επιτυγχάνει να αντικαθιστά πλήρως οποιονδήποτε εκτιμητή *scikit-learn* ([2]), για επιβλεπόμενες εργασίες μηχανικής μάθησης.

Για τους σκοπούς αυτής της διπλωματικής εργασίας, απομονώσαμε και χρησιμοποιήσαμε τον μηχανισμό μεταμάθησης του *auto-sklearn*. Στις επόμενες ενότητες, θα προσφέρουμε μία σύνοψη της μελέτης μας πάνω στον τρόπο λειτουργίας του μηχανισμού και μια λεπτομερή περιγραφή των επιμέρους τμημάτων που απαρτίζουν τον μηχανισμό.

Παρακάτω, θα εκθέσουμε μία αριθμημένη λίστα βημάτων που περιγράφει αυτόν τον μηχανισμό μεταμάθησης. Ουσιαστικά, το *auto-sklearn*:

1. **εξάγει ένα σετ μεταχαρακτηριστικών** από ένα σύνολο δεδομένων εισόδου.
2. **συγκρίνει αυτό το διάνυσμα μεταχαρακτηριστικών με έναν αριθμό άλλων διανυσμάτων μεταχαρακτηριστικών** που είναι αποθηκευμένα στη βάση δεδομένων μεταμάθησής του και βρίσκει το πιο παρόμοιο. Ουσιαστικά, κάθε ένα από αυτά τα διανύσματα μεταχαρακτηριστικών αντιστοιχεί σε ένα συγκεκριμένο σετ δεδομένων, άρα με άλλα λόγια το *auto-sklearn* βρίσκει το πιο παρόμοιο σετ δεδομένων που υπάρχει στη βάση δεδομένων μεταμάθησής του.
3. **προτείνει ένα σετ διαμορφώσεων διοχέτευσης μηχανικής μάθησης** που βαθμολογήθηκε υψηλά στο πιο παρόμοιο σετ δεδομένων. Το *auto-sklearn* κρατάει επίσης αυτές τις ρυθμίσεις στη μετάβασή του, μαζί με τα αντίστοιχα μετρικά αποτελέσματά για κάθε σετ δεδομένων. Αυτές οι προτεινόμενες ρυθμίσεις είναι βέβαιες ότι θα λειτουργούν καλά στο σετ δεδομένων εισόδου.

## 3.2 Διαμορφώσεις Διοχετεύσεων Μηχανικής Μάθησης

### 3.2.1 Επισκόπηση

Μία διαμόρφωση διοχέτευσης μηχανικής μάθησης είναι ουσιαστικά μία λεπτομερής περιγραφή μιας ολόκληρης διοχέτευσης μηχανικής μάθησης. Το πακέτο *auto-sklearn* περιλαμβάνει μία βάση δεδομένων που περιέχει μια πληθώρα τέτοιων διαμορφώσεων. Θα περιγράψουμε αυτή τη βάση δεδομένων μεταμάθησης σε επόμενη ενότητα.

Ο στόχος του *auto-sklearn* είναι να προτείνει διαμορφώσεις που είναι βέβαιες να πετύχουν υψηλή βαθμολογία για ένα δοσμένο σετ δεδομένων και εργασία μηχανικής μάθησης. Το απόσπασμα κώδικα παρακάτω δείχνει ένα παράδειγμα διαμόρφωσης μεταμάθησης για μια εργασία ταξινόμησης:

Listing 3.1: Μια διαμόρφωση μεταμάθησης για ταξινόμηση

```

1 balancing:strategy, Value: 'weighting'
2 classifier:__choice__, Value: 'libsvm_svc'
3 classifier:libsvm_svc:C, Value: 6384.641073379224
4 classifier:libsvm_svc:coef0, Value: -0.1592835134753816
5 classifier:libsvm_svc:degree, Value: 2
6 classifier:libsvm_svc:gamma, Value: 0.6866143858851854
7 classifier:libsvm_svc:kernel, Value: 'poly'
8 classifier:libsvm_svc:max_iter, Constant: -1
9 classifier:libsvm_svc:shrinking, Value: 'False'
10 classifier:libsvm_svc:tol, Value: 2.6500330000385803e-05
11 data_preprocessing:categorical_transformer:categorical_encoding:__choice__, Value: 'no_encoding'
12 data_preprocessing:categorical_transformer:category_coalescence:__choice__, Value: '
    no_coalescence'
13 data_preprocessing:numerical_transformer:imputation:strategy, Value: 'median'
14 data_preprocessing:numerical_transformer:rescaling:__choice__, Value: 'normalize'
15 feature_preprocessor:__choice__, Value: 'no_preprocessing'

```

Παρόμοια, το ακόλουθο απόσπασμα δείχνει μια πραγματική διαμόρφωση μεταμάθησης που θα μπορούσε να χρησιμοποιηθεί για μια εργασία παλινδρόμησης:

Listing 3.2: Μια διαμόρφωση μεταμάθησης για παλινδρόμηση

```

1 data_preprocessing:categorical_transformer:categorical_encoding:__choice__, Value: 'no_encoding'
2 data_preprocessing:categorical_transformer:category_coalescence:__choice__, Value: '
    no_coalescence'
3 data_preprocessing:numerical_transformer:imputation:strategy, Value: 'most_frequent'
4 data_preprocessing:numerical_transformer:rescaling:__choice__, Value: 'robust_scaler'
5 data_preprocessing:numerical_transformer:rescaling:robust_scaler:q_max, Value:
    0.8280417820125114
6 data_preprocessing:numerical_transformer:rescaling:robust_scaler:q_min, Value:
    0.0781634653874277
7 feature_preprocessor:__choice__, Value: 'kitchen_sinks'
8 feature_preprocessor:kitchen_sinks:gamma, Value: 8.438432240830361e-05
9 feature_preprocessor:kitchen_sinks:n_components, Value: 2984
10 regressor:__choice__, Value: 'ard_regression'

```

```

11 regressor:ard_regression:alpha_1, Value: 0.00044036509169446026
12 regressor:ard_regression:alpha_2, Value: 4.039147500668822e-10
13 regressor:ard_regression:fit_intercept, Constant: 'True'
14 regressor:ard_regression:lambda_1, Value: 8.922721154590444e-05
15 regressor:ard_regression:lambda_2, Value: 3.0105431227198885e-05
16 regressor:ard_regression:n_iter, Constant: 300
17 regressor:ard_regression:threshold_lambda, Value: 1899.168836704701
18 regressor:ard_regression:tol, Value: 0.011611373742389547

```

Όπως μπορεί κανείς να δει παραπάνω, μια διαμόρφωση είναι ένα αντικείμενο σαν λεξικό που ορίζει **ένα βήμα προεπεξεργασίας** και **και ένα βήμα εκτιμητή** για μια διοχέτευση μηχανικής μάθησης. Σε επόμενο κεφάλαιο, θα δούμε πώς μετατρέπουμε αυτές τις περιγραφές σε αντικείμενα *sklearn* που όντως υλοποιούν προεπεξεργαστές και εκτιμητές.

### 3.2.2 Ο Χώρος Διαμόρφωσης

Ο χώρος διαμόρφωσης αποτελείται ουσιαστικά από όλες τις διαμορφώσεις διοχετεύσεων που είναι βάσιμες για μια δεδομένη εργασία μηχανικής μάθησης. Για παράδειγμα, στην περίπτωση μιας εργασίας *ταξινόμησης*, ο χώρος διαμόρφωσης θα περιέχει διαμορφώσεις διοχετεύσεων που έχουν μόνο *ταξινομητές* σαν εκτιμητές. Παρόμοια, στην περίπτωση μιας εργασίας *παλιωδρόμησης*, ο χώρος διαμόρφωσης θα περιέχει διαμορφώσεις διοχετεύσεων που έχουν μόνο *παλιωδρομητές*.

Ωστόσο, ο τύπος της δοσμένης εργασίας μηχανικής μάθησης δεν είναι το μόνο κριτήριο που επηρεάζει τη δομή του χώρου διαμόρφωσης. Πιο συγκεκριμένα, η δομή του χώρου διαμόρφωσης επηρεάζεται από τρεις παραμέτρους:

1. **Τον τύπο της εργασίας μηχανικής μάθησης.**
2. **Εάν το σετ δεδομένων εισόδου είναι αραιό ή όχι.**
3. **Εάν το σετ δεδομένων εισόδου περιέχει χαμένες τιμές ή όχι.**

Σε επόμενη υποενότητα, θα αναφερθούμε στην κλάση *XYDataManager*, η οποία μας βοηθάει να υπολογίσουμε όλες τις προαναφερθείσες παραμέτρους. Ο χώρος διαμόρφωσης είναι ένα από τα πρώτα πράγματα που υπολογίζει ο μηχανισμός μεταμάθησης του *auto-sklearn*, προτού παράξει τις προτεινόμενες διαμορφώσεις διοχετεύσεων.

#### 3.2.2.1 Η Συνάρτηση *get\_configuration\_space*

Το πακέτο *auto-sklearn* εκθέτει τη συνάρτηση *get\_configuration\_space* του API η οποία παράγει ένα αντικείμενο χώρου διαμόρφωσης για μια δεδομένη εργασία μηχανικής μάθησης. Ορίστε η υπογραφή της συνάρτησης *get\_configuration\_space* του API:

- **Ορίσματα Εισόδου:**

- **info:** Ένα λεξικό που περιέχει τις απαραίτητες πληροφορίες για τον υπολογισμό του χώρου διαμόρφωσης. Αυτό το λεξικό πρέπει να περιέχει:
  - \* **task:** Έναν ακέραιο που αντιστοιχεί στον τύπο της εργασίας μηχανικής μάθησης.

- \* **is\_sparse**: Μια τιμή *Μπουλ* που περιγράφει εάν το σει δεδομένων εισόδου είναι αραιό ή όχι.
- \* **has\_missing**: Μια τιμή *Μπουλ* που περιγράφει εάν υπάρχουν χαμένες ή άπειρες τιμές στο σει δεδομένων.
- **include\_estimators**: Μια λίστα με τα ονόματα των εκτιμητών που θα χρησιμοποιηθούν, αποκλειστικά. Αυτή τίθεται από προεπιλογή ως *None*, οπότε συμπεριλαμβάνονται όλοι οι εκτιμητές, από προεπιλογή.
- **exclude\_estimators**: Μια λίστα με τα ονόματα των εκτιμητών που θα εξαιρεθούν. Αυτή τίθεται από προεπιλογή ως *None*, οπότε κανένας εκτιμητής δεν αποκλείεται.
- **include\_preprocessors**: Μια λίστα με τα ονόματα των προεπεξεργασιών που θα χρησιμοποιηθούν, αποκλειστικά. Αυτή τίθεται από προεπιλογή ως *None*, οπότε συμπεριλαμβάνονται όλοι οι προεπεξεργαστές, από προεπιλογή.
- **exclude\_preprocessors**: Μια λίστα με τα ονόματα των προεπεξεργασιών που θα εξαιρεθούν. Αυτή τίθεται από προεπιλογή ως *None*, οπότε κανένας προεπεξεργαστής δεν αποκλείεται, από προεπιλογή.

- **Τιμή Επιστροφής:**

- **Το αντικείμενο χώρου διαμόρφωσης**

Σε επόμενη ενότητα, θα μιλήσουμε για το *XYDataManager*, ένα αντικείμενο κλάσης που παρέχεται από το *auto-sklearn*, το οποίο υπολογίζει το λεξικό **info** για εμάς.

### 3.2.3 Ο Κατάλογος Μετα-Δεδομένων

Ο κατάλογος μεταδεδομένων του *auto-sklearn* είναι ένας πραγματικός κατάλογος εντός του πακέτου *auto-sklearn* όπου είναι αποθηκευμένη όλη η μετα-γνώση του *auto-sklearn*. Αυτή η μετα-γνώση οργανώνεται κατ' ακολουθία με το φορμάτ *ASlib* ([30]). Αυτό σημαίνει ότι τα μεταδεδομένα κατηγοριοποιούνται σε υποκαταλόγους, ανάλογα με τρεις παράγοντες:

1. **Η μετρική συνάρτηση**
2. **Ο τύπος της εργασίας**
3. **Η αραιότητα του σει δεδομένων**

Έτσι, για να δώσουμε ένα παράδειγμα, εάν φανταστούμε ένα πυκνό σει δεδομένων για μια εργασία πολυταξικής ταξινόμησης, με την ακρίβεια ως μετρικό, τότε αυτό θα ενέπιπτε στον υποκατάλογο *accuracy\_multiclass.classification\_dense* του καταλόγου μεταδεδομένων του *auto-sklearn*.

Παρόμοια, στην περίπτωση ενός αραιού σει δεδομένων σε μια εργασία παλινδρόμησης, με το μέσο απόλυτο σφάλμα ως μετρική, ο υποκατάλογος είναι: *mean\_absolute\_error\_regression\_dense*



### 3.2.4 Η Κλάση *XYDataManager*

Η κλάση *XYDataManager* που παρέχεται από το *auto-sklearn* εξάγει και αποθηκεύει πληροφορίες που είναι απαραίτητες ώστε να παραχθούν προτεινόμενες διαμορφώσεις για μία δοσμένη εργασία μηχανικής μάθησης. Εδώ είναι τα ορίσματα εισόδου για τον κατασκευαστή *XYDataManager*:

- **X**: Ένας πίνακας που περιλαμβάνει τα δείγματα του σετ δεδομένων εκπαίδευσης.
- **y**: Ένας πίνακας που περιλαμβάνει τους στόχους του σετ δεδομένων εκπαίδευσης.
- **X\_test**: Ένας πίνακας που περιλαμβάνει τα δείγματα του σετ δεδομένων δοκιμής.
- **y\_test**: Ένας πίνακας που περιλαμβάνει τους στόχους του σετ δεδομένων δοκιμής.
- **task**: Ένας ακέραιος που αντιστοιχεί στον τύπο της εργασίας μηχανικής μάθησης. Για τους σκοπούς της διεργασίας του Kale-AutoML, αυτός ο ακέραιος θα είναι είτε *1*, είτε *2* είτε *4* που αντιστοιχούν σε *παλινδρόμηση*, *δυαδική ταξινόμηση* και *πολυταξική ταξινόμηση* αντίστοιχα.
- **feat\_types**: Μια λίστα αλφαριθμητικών που περιγράφει τον τύπο κάθε χαρακτηριστικού. Η λίστα έχει μήκος ίσο με τον αριθμό των χαρακτηριστικών στο σετ δεδομένων. Κάθε στοιχείο της λίστας μπορεί να είναι είτε «*κατηγορηματικό*» είτε «*αριθμητικό*»<sup>1</sup>.
- **dataset\_name**: Το όνομα του σετ δεδομένων.

Το ακόλουθο απόσπασμα κώδικα δείχνει τον κώδικα της συνάρτησης κατασκευαστή του *XYDataManager*:

Listing 3.3: Η συνάρτηση κατασκευαστή του *XYDataManager*

```

1  class XYDataManager(AbstractDataManager):
2
3      def __init__(
4          self,
5          X: np.ndarray,
6          y: np.ndarray,
7          X_test: Optional[np.ndarray],
8          y_test: Optional[np.ndarray],
9          task: int,
10         feat_type: List[str],
11         dataset_name: str
12     ):
13         super(XYDataManager, self).__init__(dataset_name)
14
15         self.info['task'] = task
16         if sparse.issparse(X):
17             self.info['is_sparse'] = 1
18             self.info['has_missing'] = np.all(np.isfinite(X.data))

```

<sup>1</sup>Η τιμή ενός κατηγορηματικού χαρακτηριστικού δηλώνει ότι ένα δείγμα ανήκει σε μια συγκεκριμένη κατηγορία. Αυτοί οι τύποι χαρακτηριστικών έχουν ένα πεπερασμένο αριθμό υποψήφιας τιμών. Τα αριθμητικά χαρακτηριστικά από την άλλη, εκφράζουν ποσοτικά χαρακτηριστικά ενός δείγματος. Ένα παράδειγμα κατηγορηματικού χαρακτηριστικού είναι η χώρα γέννησης ενός ατόμου, ενώ ένα αριθμητικό χαρακτηριστικό είναι το ύψος του ατόμου.

```

19     else:
20         self.info['is_sparse'] = 0
21         self.info['has_missing'] = np.all(np.isfinite(X))
22
23     label_num = {
24         REGRESSION: 1,
25         BINARY_CLASSIFICATION: 2,
26         MULTIOUTPUT_REGRESSION: y.shape[-1],
27         MULTICLASS_CLASSIFICATION: len(np.unique(y)),
28         MULTILABEL_CLASSIFICATION: y.shape[-1]
29     }
30
31     self.info['label_num'] = label_num[task]
32
33     self.data['X_train'] = X
34     self.data['Y_train'] = y
35     if X_test is not None:
36         self.data['X_test'] = X_test
37     if y_test is not None:
38         self.data['Y_test'] = y_test
39
40     if feat_type is not None:
41         for feat in feat_type:
42             allowed_types = ['numerical', 'categorical']
43             if feat.lower() not in allowed_types:
44                 raise ValueError("Entry '%s' in feat_type not in %s" %
45                                 (feat.lower(), str(allowed_types)))
46
47     self.feat_type = feat_type
48
49     # TODO: try to guess task type!
50
51     if len(y.shape) > 2:
52         raise ValueError('y must not have more than two dimensions, '
53                          'but has %d.' % len(y.shape))
54
55     if X.shape[0] != y.shape[0]:
56         raise ValueError('X and y must have the same number of '
57                          'datapoints, but have %d and %d.' % (X.shape[0],
58                                                                  y.shape[0]))
59
60     if self.feat_type is None:
61         self.feat_type = ['Numerical'] * X.shape[1]
62     if X.shape[1] != len(self.feat_type):
63         raise ValueError('X and feat_type must have the same number of columns, '
64                          'but are %d and %d.' %
65                          (X.shape[1], len(self.feat_type)))

```

Κατά την αρχικοποίησή του, ένα αντικείμενο *XYDataManager* εξάγει πληροφορίες που

είναι απαραίτητες για την διεργασία μεταμάθησης και τις αποθηκεύει στο πεδίο κλάσης **info**. Αυτό το πεδίο είναι ουσιαστικά ένα λεξικό Python που περιλαμβάνει τα ακόλουθα πεδία:

- **task**: Ένας ακέραιος που αντιστοιχεί στον τύπο της εργασίας μηχανικής μάθησης.
- **is\_sparse**: Μια τιμή *Μπουλ* που περιγράφει αν το σετ εισόδου είναι αραιό ή όχι. Για να αποφασίσει επ' αυτού, το *XYDataManager* χρησιμοποιεί τη συνάρτηση **issparse** ([31]) το δομοστοιχείο **sparse** του **scipy** ([14]).
- **has\_missing**: Μια τιμή *Μπουλ* που περιγράφει εάν υπάρχουν χαμένες ή άπειρες τιμές στο σετ δεδομένων.

Όλα τα πεδία που περιγράψαμε παραπάνω είναι απαραίτητα για την εύρεση:

1. του σωστού **καταλόγου μεταδεδομένων** για μια εργασία μηχανικής μάθησης. Σε επόμενη ενότητα, θα περιγράψουμε πώς το *auto-sklearn* χωρίζει τη μετα-βάση του σε καταλόγους και πώς βρίσκει το σωστό κατάλογο μεταδεδομένων για μία δεδομένη εργασία μηχανικής μάθησης.
2. του **χώρου διαμόρφωσης** στον οποίο θα ψάξει το *auto-sklearn* για υποψήφιες διαμορφώσεις μεταμάθησης. Θα περιγράψουμε πώς το *auto-sklearn* βρίσκει τον χώρο διαμόρφωσης για ένα δεδομένο πείραμα σε ακόλουθη ενότητα.

### 3.2.5 Οι Κλάσεις *SimpleRegressionPipeline* και *SimpleClassificationPipeline*

Αυτές οι δύο κλάσεις υλοποιούν την εργασία ταξινόμησης. Υλοποιούν μία διοχέτευση, που περιλαμβάνει βήματα προεπεξεργασίας και ένα βήμα εκτιμητή στο τέλος.

Ένα αντικείμενο αυτών των κλάσεων αρχικοποιείται περνώντας ένα αντικείμενο διαμόρφωσης μεταμάθησης (3.2) στον κατασκευαστή της αντίστοιχης κλάσης. Μετά από αυτό, μπορεί κανείς να καλέσει:

1. τη μέθοδο **fit** του αντικειμένου για να εφαρμόσει την διοχέτευση σε ένα σετ δεδομένων εκπαίδευσης.
2. τη μέθοδο **predict** του αντικειμένου για να κάνει προβλέψεις σε ένα σετ δεδομένων δοκιμής.

## 3.3 Μετα-Χαρακτηριστικά

### 3.3.1 Επισκόπηση

Σε αυτή την ενότητα, θα περιγράψουμε τις βασικές ιδέες του υποκείμενου μηχανισμού εξαγωγής μεταχαρακτηριστικών που χρησιμοποιεί το *auto-sklearn* ([1]). Αυτό είναι ένα από τα βασικά κομμάτια το πυρήνα μεταμάθησης που χρησιμοποιήσαμε για την διανεμημένη AutoML διεργασία μας.

Το *Auto-sklearn* χωρίζει το σετ των μεταχαρακτηριστικών που μπορεί να υπολογίσει σε δύο κύριες κατηγορίες:

1. **Απλά Μεταχαρακτηριστικά**: Αυτά τα μεταχαρακτηριστικά είναι υπολογιστικά φθηνά, και δεν απαιτούν μετασχηματισμούς στο σετ δεδομένων ώστε να υπολογιστούν.

2. **1HotEncoded Μεταχαρακτηριστικά:** Αυτά τα μεταχαρακτηριστικά είναι πιο ακριβά υπολογιστικά, και υπολογίζονται χρησιμοποιώντας τη μήτρα χαρακτηριστικών **1HotEncoded** του σετ δεδομένων.

Θα αναλύσουμε αυτές τις δύο κύριες κατηγορίες μεταχαρακτηριστικών στις ακόλουθες υποενότητες.

### 3.3.2 Απλά Μετα-Χαρακτηριστικά

Αυτά τα μετα-χαρακτηριστικά εξάγονται απευθείας από το Dataset εισόδου, χωρίς προηγούμενους μετασχηματισμούς ή προεπεξεργασία, οπότε είναι υπολογιστικά φθηνά, γενικά. Εδώ είναι η πλήρης λίστα απλών μετα-χαρακτηριστικών που μπορεί να υπολογίσει το *auto-sklearn*:

- **Αριθμός αντικειμένων**
- **Λογαριθμικός αριθμός αντικειμένων**
- **Αριθμός κλάσεων**
- **Αριθμός χαρακτηριστικών**
- **Λογαριθμικός αριθμός χαρακτηριστικών**
- **Αριθμός χαρακτηριστικών με χαμένες τιμές**
- **Εάν λείπουν τιμές ή όχι**
- **Αριθμός αντικειμένων με χαμένες τιμές**
- **Ποσοστό αντικειμένων με χαμένες τιμές**
- **Αριθμός χαρακτηριστικών με χαμένες τιμές**
- **Ποσοστό χαρακτηριστικών με χαμένες τιμές**
- **Αριθμός χαμένων τιμών**
- **Ποσοστό χαμένων τιμών**
- **Αριθμός αριθμητικών χαρακτηριστικών**
- **Αριθμός κατηγορηματικών χαρακτηριστικών**
- **Αναλογία αριθμητικών προς κατηγορηματικών χαρακτηριστικών**
- **Αναλογία κατηγορηματικών προς αριθμητικών χαρακτηριστικών**
- **Αναλογία χαρακτηριστικών προς αντικειμένων**
- **Λογαριθμική αναλογία χαρακτηριστικών προς αντικειμένων**
- **Αναλογία αντικειμένων προς χαρακτηριστικών**
- **Λογαριθμική αναλογία αντικειμένων προς χαρακτηριστικών**
- **Αριθμός εμφανίσεων κάθε κλάσης**
- **Ελάχιστη πιθανότητα κλάσης**
- **Μέγιστη πιθανότητα κλάσης**
- **Μέση τιμή της πιθανότητας κλάσης**
- **Τυπική απόκλιση της πιθανότητας κλάσης**

- Αριθμός συμβόλων<sup>2</sup>
- Ελάχιστος αριθμός συμβόλων
- Μέγιστος αριθμός συμβόλων
- Τυπική απόκλιση του αριθμού συμβόλων
- Άθροισμα των αριθμών συμβόλων
- Εντροπία κλάσης

### 3.3.3 Μετα-Χαρακτηριστικά 1HotEncoded

Ούτως ώστε να εξάγει αυτά τα μετα-χαρακτηριστικά, το Dataset εισόδου υποβάλλεται σε ένα μετασχηματισμό. Πιο συγκεκριμένα, δημιουργείται μία μήτρα χαρακτηριστικών **1HotEncoded**. Αυτά τα μετα-χαρακτηριστικά εξάγονται στην πραγματικότητα από αυτή τη μήτρα χαρακτηριστικών 1HotEncoded, όχι από το ίδιο το Dataset. Γενικά, ο υπολογισμός αυτών των χαρακτηριστικών είναι υπολογιστικά ακριβός, αφού αυτά τα χαρακτηριστικά πιο προσανατολισμένα προς την επιστήμη δεδομένων από τα απλά μετα-χαρακτηριστικά που καταγράψαμε παραπάνω. Εδώ είναι η πλήρης λίστα των μετα-χαρακτηριστικών 1HotEncoded που μπορεί να υπολογίσει το `auto-sklearn`:

- Λοξότητες
- Ελάχιστη λοξότητα
- Μέγιστη λοξότητα
- Μέση λοξότητα
- Τυπική απόκλιση λοξότητας
- Κυρτώσεις
- Ελάχιστη κύρτωση
- Μέγιστη κύρτωση
- Μέση κύρτωση
- Τυπική απόκλιση κύρτωσης

Τώρα που η γνώση μας των τύπων μεταδεδομένων που υποστηρίζει το `auto-sklearn` έχει επεκταθεί, είμαστε έτοιμοι να εξερευνήσουμε πώς μπορεί κανείς να χρησιμοποιήσει το `auto-sklearn` ώστε πράγματι να υπολογίσει αυτά τα μετα-χαρακτηριστικά από ένα δοσμένο σετ δεδομένων εισόδου.

<sup>2</sup>Ο όρος «σύμβολο» εκφράζει την τιμή που μπορεί να έχει ένα κατηγορηματικό χαρακτηριστικό.

### 3.3.4 Συναρτήσεις API για Εξαγωγή Μετα-Χαρακτηριστικών

Για να υπολογίσει τα μετα-χαρακτηριστικά που ανήκουν στις δύο κατηγορίες που αναλύσαμε παραπάνω, το `auto-sklearn` προσφέρει δύο κύριες συναρτήσεις API:

1. **`calculate_all_metafeatures_with_labels`**
2. **`calculate_all_metafeatures_encoded_labels`**

Και οι δύο αυτές συναρτήσεις λαμβάνουν τις ίδιες παραμέτρους ως εισόδους. Ας δούμε αυτές τις παραμέτρους λεπτομερώς:

- **`X`**: Τα δείγματα του σετ δεδομένων εκπαίδευσης.
- **`y`**: Οι στόχοι του σετ δεδομένων εκπαίδευσης.
- **`categorical`**: Μια λίστα Boolean τιμών που έχει μήκος ίσο με τον αριθμό χαρακτηριστικών σε κάθε δείγμα. Αν ένα στοιχείο στη λίστα είναι `True`, τότε το αντίστοιχο χαρακτηριστικό θεωρείται ως κατηγορηματικό χαρακτηριστικό. Ειδικά, είναι ένα αριθμητικό χαρακτηριστικό.
- **`dataset_name`**: Το όνομα του σετ δεδομένων.
- **`dont_calculate`**: Ένα σετ μετα-χαρακτηριστικών που δεν θα πρέπει να υπολογιστούν για το συγκεκριμένο σετ δεδομένων εισόδου και εργασία μηχανικής μάθησης.

Η παράμετρος `dont_calculate` που αναφέραμε παραπάνω χρησιμοποιείται κατά κύριο λόγο για να αποκλείσει μετα-χαρακτηριστικά στις περιπτώσεις εργασιών παλινδρόμησης. Πιο συγκεκριμένα, εφόσον οι εργασίες παλινδρόμησης δεν έχουν κλάσεις ως στόχους, τα ακόλουθα μεταχαρακτηριστικά πρέπει να αποκλειστούν:

- **Αριθμός κλάσεων**
- **Αριθμός εμφανίσεων κάθε κλάσης**
- **Ελάχιστη πιθανότητα κλάσης**
- **Μέγιστη πιθανότητα κλάσης**
- **Μέση τιμή της πιθανότητας κλάσης**
- **Τυπική απόκλιση της πιθανότητας κλάσης**
- **Εντροπία κλάσης**

Και οι δύο αυτές συναρτήσεις επιστρέφουν ένα αντικείμενο `DatasetMetafeatures` από το δομοστοιχείο `metalearning.metafeatures` του `auto-sklearn`. Αυτό το αντικείμενο ουσιαστικά κρατά ένα λεξικό των υπολογισμένων μεταχαρακτηριστικών στο πεδίο του `metafeature_values`.

### 3.3.5 Η Κλάση `MetaBase`

Το αντικείμενο κλάσης `MetaBase` είναι ένα container για μεταδεδομένα σετ δεδομένων (τιμές μεταχαρακτηριστικών), διαμορφώσεις διοχετεύσεων και αποτελέσματα πειραμάτων. Είναι ουσιαστικά ένα περιτύλιγμα γύρω από τη μεταγνώση του `auto-sklearn`, που αποθηκεύεται στον κατάλογο μεταδεδομένων που περιγράψαμε προηγουμένως.

Σε ένα αντικείμενο *MetaBase*, το *auto-sklearn* αποθηκεύει τα μεταχαρακτηριστικά ενός σετ δεδομένων, καθώς επίσης και τα αποτελέσματα επικύρωσης διαφόρων διαμορφώσεων διοχετεύσεων για τη συγκεκριμένη εργασία μηχανικής μάθησης.

Για να κατασκευάσουμε ένα αντικείμενο *MetaBase*, πρέπει να προμηθεύσουμε:

- **ένα χώρο διαμόρφωσης**
- **έναν κατάλογο μεταδεδομένων**

Επιπλέον, αφού αρχικοποιήσουμε ένα αντικείμενο *MetaBase* μπορούμε να προσθέσουμε καταχώρηση ενός σετ δεδομένων χρησιμοποιώντας τη μέθοδο *add\_dataset*. Αυτή η μέθοδος απαιτεί:

- **το όνομα του σετ δεδομένων**
- **ένα αντικείμενο *DatasetMetafeatures*** που περιέχει τις τιμές μεταχαρακτηριστικών για το αντίστοιχο σετ δεδομένων

Η προσθήκη ενός σετ δεδομένων εισόδου στο αντικείμενο *MetaBase* είναι απαραίτητη για τον υπολογισμό των προτεινόμενων διαμορφώσεων για αυτό το σετ δεδομένων χρησιμοποιώντας τη συνάρτηση *suggest\_via\_metalearning*, την οποία θα αναλύσουμε σε επόμενη ενότητα.

### 3.4 Η Συνάρτηση *suggest\_via\_metalearning*

Η συνάρτηση *suggest\_via\_metalearning* του δομοστοιχείου *autosklearn.metalearning.mismbo* είναι μία από τις σημαντικότερες συναρτήσεις του API στο μηχανισμό μεταμάθησης του *auto-sklearn*, αφού είναι αυτή που παράγει τις προτεινόμενες διαμορφώσεις διοχέτευσης για ένα σετ δεδομένων εισόδου και μία εργασία μηχανικής μάθησης. Ας δούμε πιο αναλυτικά την υπογραφή της συνάρτησης:

- **Ορίσματα Εισόδου:**

- **meta\_base:** Ένα αντικείμενο *MetaBase* αρχικοποιημένο με το σωστό κατάλογο μεταδεδομένων και το χώρο διαμόρφωσης στον οποίο η *suggest\_via\_metalearning* θα αναζητήσει για την προτεινόμενες διαμορφώσεις.
- **dataset\_name:** Το όνομα του σετ δεδομένων για το οποίο θα χρησιμοποιηθούν οι προτεινόμενες διαμορφώσεις. Το σετ δεδομένων πρέπει να προστεθεί στο αντικείμενο *MetaBase*. Αυτό μπορεί να γίνει με τη μέθοδο *MetaBase.add\_dataset* που επιδείξαμε στην υποενότητα [subsection 3.3.5](#).
- **metric:** Η μετρική συνάρτηση.
- **task:** Ένας ακέραιος που αντιστοιχεί στον τύπο της εργασίας μηχανικής μάθησης.
- **sparse:** Μία *Μπουλ* που περιγράφει εάν το σετ δεδομένων εισόδου είναι αραιό ή όχι.
- **num\_initial\_configurations:** Ο αριθμός των προτεινόμενων διαμορφώσεων που θα παραχθούν.

- **Τιμή Επιστροφής:**

- **configurations**: Μια λίστα με τις προτεινόμενες διαμορφώσεις.

Σε ένα ακόλουθο κεφάλαιο, θα περιγράψουμε πώς οπίσθιο τμήμα μηχανικής μάθησης του Kale χρησιμοποιεί την *suggest\_via\_metalearning* για να παράξει τις προτεινόμενες διαμορφώσεις που αργότερα θα μετατραπούν σε πραγματικές διοχετεύσεις KFP.

## 3.5 Βοηθητικές Συναρτήσεις και Κλάσεις

### 3.5.1 Η Κλάση *InputValidator*

Η *InputValidator* που παρέχεται από το δομοστοιχείο *autosklearn.data.validation* είναι μία χρηστική κλάση που μπορεί να χρησιμοποιηθεί για να βεβαιώσει ότι το σετ δεδομένων εισόδου συμμορφώνεται στις απαιτήσεις του *auto-sklearn*. Η βασική μέθοδος του API της *InputValidator* ονομάζεται *validate* και η υπογραφή της παρουσιάζεται παρακάτω:

- **Ορίσματα Εισόδου:**

- **X**: Τα δείγματα ενός σετ δεδομένων.
- **y**: Οι στόχοι ενός σετ δεδομένων.
- **is\_classification**: Μία boolean τιμή που εκφράζει εάν η εργασία για την οποία θα χρησιμοποιηθεί το σετ δεδομένων εισόδου είναι ταξινόμηση ή όχι.

- **Τιμές Επιστροφής:**

- **X**: Τα επικυρωμένα δείγματα.
- **y**: Οι επικυρωμένοι στόχοι.

Η *InputValidator.validate* ουσιαστικά υλοποιεί δύο λειτουργικότητες:

1. Ελέγχει ότι ο αριθμός δειγμάτων αντιστοιχεί στον αριθμό στόχων στο σετ δεδομένων.
2. Ελέγχει ότι το σετ δεδομένων αποτελείται μόνο από αριθμητικά δεδομένα.
3. Ανάλογα με τον τύπο της εργασίας, που εκφράζεται από το όρισμα εισόδου *is\_classification*, ελέγχει ότι οι στόχοι του σετ δεδομένων μπορούν να χρησιμοποιηθούν για αυτή την εργασία (ταξινόμηση ή παλινδρόμηση).



# Η Προσέγγιση μας

---

Όπως εξηγήσαμε σε προηγούμενο κεφάλαιο, τα πειράματα AutoML περιέχουν βήματα τα οποία μπορούν να παραλληλοποιηθούν. Σε έναν τοπικό υπολογιστή, μόλις ο μηχανισμός μετα-μάθησης του *auto-sklearn* υπολογίσει τις προτεινόμενες διαμορφώσεις διοχέτευσης, το *auto-sklearn* εκπαιδεύει τις αντίστοιχες διοχετεύσεις μηχανικής μάθησης τοπικά, σαν παράλληλες διεργασίες, έτσι ώστε η διοχέτευση που με την καλύτερη απόδοση να βρεθεί και να επιστραφεί στον χρήστη. Ο στόχος μας ήταν να μεταφέρουμε ολόκληρη αυτή την διαδικασία στον *Κυβερνήτη*, εκμεταλλευόμενοι τον μηχανισμό ενορχήστρωσης διοχετεύσεων μηχανικής μάθησης του *Kale*, έτσι ώστε να εκπαιδεύουμε μοντέλα μηχανικής μάθησης ως παράλληλες διοχετεύσεις στο *Kubeflow*. Δημιουργήσαμε έναν μηχανισμό που επιτρέπει την εκτέλεση πειραμάτων AutoML αποδοτικά, και κατανεμημένα, στον *Κυβερνήτη*.

Ας περιγράψουμε τα βήματα της διεργασίας του *Kale* για πειράματα AutoML:

1. Ο χρήστης παρέχει ένα **σύνολο δεδομένων** και τον **τύπο της εργασίας μηχανικής μάθησης** (κατηγοριοποίηση ή παλινδρόμησης) ως είσοδο στην συνάρτηση `run_automl()` του *Kale*.
2. **Το Kale δημιουργεί μία διοχέτευση του Kubeflow**. Η διοχέτευση αυτή ονομάζεται *Ενορχηστρωτής* και θα αναφερόμαστε σε αυτή με αυτό το όνομα για το υπόλοιπο του κεφαλαίου.
3. **Το Kale επιστρέφει ένα αντικείμενο *AutoMLExperiment* στον χρήστη** (τιμή επιστροφής της συνάρτησης `run_automl()`). Το αντικείμενο αυτό ουσιαστικά θα είναι ένα εργαλείο παρακολούθησης της κατάστασης ολόκληρης της διαδικασίας AutoML.
4. Χρησιμοποιώντας τον μηχανισμό μετα-μάθησης του *auto-sklearn*, **ο Ενορχηστρωτής υπολογίζει μία λίστα με προτεινόμενες διαμορφώσεις διοχετεύσεων μηχανικής μάθησης** για το σύνολο δεδομένων και τύπο εργασίας μηχανικής μάθησης που ο χρήστης έδωσε ως είσοδο (στην συνάρτηση `run_automl()`). Κάθε μια από αυτές τις διαμορφώσεις ουσιαστικά **περιγράφει μία ολόκληρη διοχέτευση μηχανικής μάθησης**.
5. **Για κάθε προτεινόμενη διαμόρφωση διοχέτευσης, ο Ενορχηστρωτής δημιουργεί μια νέα διοχέτευση του Kubeflow**. Οι διοχετεύσεις αυτές ονομάζονται *Ροές Διαμόρφωσης* και θα αναφερόμαστε σε αυτές χρησιμοποιώντας αυτό το όνομα για το υπόλοιπο του κεφαλαίου. Κάθε μία από αυτές τις *Ροές Διαμόρφωσης* υλοποιεί την διοχέτευση μηχανικής μάθησης που η αντίστοιχη διαμόρφωση διοχέτευσης περιγράφει.

6. **Οι Ροές Διαμόρφωσης τρέχουν παράλληλα**, περνώντας το σύνολο δεδομένων από ένα στάδιο προ-επεξεργασίας, εκπαιδεύοντας το μοντέλο, και παράγοντας αποτελέσματα δοκιμαστικού σετ **ενώ ο Ενορχηστρωτής ελέγχει την εξέλιξη τους**.
7. Μόλις όλες οι Ροές Διαμόρφωσης ολοκληρώσουν την λειτουργία τους, **ο Ενορχηστρωτής συγκεντρώνει τα αποτελέσματα τους και με βάση αυτά επιλέγει την καλύτερη Ροή Διαμόρφωσης**.
8. **Ο Ενορχηστρωτής δημιουργεί ένα πείραμα Katib έτσι ώστε να βελτιστοποιηθεί περαιτέρω το εκπαιδευμένο μοντέλο** της Ροής Διαμόρφωσης με το μεγαλύτερο σκορ.
9. **Το Kale αποθηκεύει το εκπαιδευμένο και βελτιστοποιημένο μοντέλο και τραβάει ένα πλήρως αναπαραγωγίμο στιγμιότυπο Rok (2.4.5.1) του λογικού δίσκου που το περιέχει** έτσι ώστε ο χρήστης να μπορεί να έχει πρόσβαση στο μοντέλο αυτό αργότερα.

Το Kale τραβάει στιγμιότυπα λογικών δίσκων, όχι μόνο στο τέλος, αλλά σε κάθε βήμα του Ενορχηστρωτή και των Ροών Διαμόρφωσης παρέχοντας έναν βολικό τρόπο να ανακτηθούν τα αποθηκευμένα εκπαιδευμένα μοντέλα που παράχθηκαν κατά την διάρκεια της διεργασίας AutoML.

**Αυτή η διπλωματική εργασία εστιάζει κυρίως στα βήματα από 3 έως και 7 της διεργασίας AutoML του Kale.** Παρόλα αυτά, θα παράσχουμε μια επαρκή ανάλυση της λειτουργικότητας και του υπολοίπου μηχανισμού.

## 4.1 Η Συνάρτηση `run_automl`

Σε αυτή την ενότητα, θα περιγράψουμε την μορφή της συνάρτησης διεπαφής του Kale για πειράματα AutoML. Ουσιαστικά, οι χρήστες θα τρέχουν πειράματα AutoML στο KubeFlow, με μια μόνο κλήση συνάρτησης Python. Αυτή η συνάρτηση, όπως αναφέραμε προηγουμένως, ονομάζεται `run_automl` και είναι ουσιαστικά ένα σημείο εισόδου για τον μηχανισμό AutoML.

Listing 4.1: Η συνάρτηση διεπαφής `run_automl` για δημιουργία πειραμάτων AutoML με το Kale

```

1  def run_automl(
2      dataset: Dataset, task: MLTask, metric: Callable,
3      number_of_configurations: int = 5,
4      max_parallel_configurations: int = 3,
5      tuner: Optional[katib.V1beta1ExperimentSpec] = None
6  ) -> AutoMLExperiment:
7      """Runs an AutoML pipeline to find the best model for the input dataset.
8
9      [... Explain how the AutoML process works ...]
10
11     Args:
12         dataset (common.artifacts.Dataset): The input dataset for the ML task
13         task (types.MLTask): One of kale.ml.Task
14         metric (Callable): A callable object with the following call signature
15
16         >>> class my_metric:
17             >>> def __call__(self, target, x_test):
18                 >>> return self._compute_my_metric_value(target, x_test)
19
20         The name of the logged metric will be 'metric.name', if the object
21         has such attribute. Otherwise 'metric.__name__'.
22         (Auto)SKLearn metrics are supported, example:
23
24         >>> from autosklearn.metrics import accuracy
25
26         To log a different metric name from the input function name
27         ('foo.__name__'), do:
28
29         >>> foo.name = "<custom_name>"
30
31         number_of_configurations (int): The N-best configurations to run
32         (defaults to 5)
33         max_parallel_configurations (int): The maximum number of Configuration
34         Runs to run in parallel (defaults to 3)
35         tuner (katib.V1beta1ExperimentSpec): Provide a Katib spec to run HP
36         Tuning over the best performing configuration.
37
38         Cannot set algorithm and parameters. To set objective
39         configuration, don't set metric name:

```

```

40
41     >>> katib.V1beta1ExperimentSpec(
42     >>>     objective=katib.V1beta1ObjectiveSpec(
43     >>>         goal=0.99,
44     >>>         type="maximize")
45
46     Returns: An AutoMLExperiment object to track the state of the experiment.
47     """
48     if tuner:
49         if tuner.algorithm or tuner.parameters:
50             raise ValueError("Tuner: Cannot specify 'algorithm', or"
51                               " 'parameters', during an AutoML experiment")
52         if tuner.objective:
53             if (tuner.objective.objective_metric_name
54                 or tuner.objective.additional_metric_names):
55                 raise ValueError("Cannot specify metric name when running"
56                                   " AutoML experiment")
57
58     pipeline_name = "automl-orchestrate"
59     utils.rm_r(ML_ASSETS_DIR)
60     marshal.set_data_dir(ML_ASSETS_DIR)
61
62     variables = ["dataset", "task", "metric", "number_of_configurations",
63                 "max_parallel_configurations"]
64     if tuner:
65         variables.append("tuner")
66     for v in variables:
67         marshal.save(locals()[v], v)
68
69     volumes = rokutils.interactive_snapshot_and_get_volumes()
70     pipeline_config = PipelineConfig(
71         pipeline_name=pipeline_name,
72         experiment_name=_auto_ml_experiment_name(),
73         volumes=volumes)
74     pipeline = PythonProcessor(automl_orchestrate, pipeline_config).run()
75     pipeline.input_pipeline_parameters["hp_tune"] = ("true" if tuner
76                                                     else "false")
77     run = Compiler(pipeline).compile_and_run()
78     return AutoMLExperiment(run.id)

```

Οι χρήστες μπορούν να εισάγουν την συνάρτηση `run_automl()` με μία απλή εντολή `import`: » `from kale.ml import run_automl`

Ας δώσουμε μια λεπτομερή εξήγηση των παραμέτρων εισόδου της `run_automl()`:

- **dataset**: Το σύνολο δεδομένων εισόδου για το πείραμα AutoML. Αυτό πρέπει να είναι ένα αντικείμενο `Dataset` του `Kale` που να περιέχει ολόκληρο το σύνολο δεδομένων μηχανικής μάθησης του χρήστη.

- **task:** Ο τύπος της εργασίας εποπτευόμενης μηχανικής μάθησης. Αυτό μπορεί να είναι ένα από τα παρακάτω:
  1. δυαδική κατηγοριοποίηση
  2. κατηγοριοποίηση πολλαπλών τάξεων
  3. απλή οπισθοδρόμηση
- **metric:** Μια κλητή συνάρτηση μετρικής η οποία θα χρησιμοποιηθεί για να αξιολογηθεί το εκπαιδευμένο μοντέλο που θα παραχθεί από την κάθε Ροή Διαμόρφωσης.
- **number\_of\_configurations:** Ο αριθμός των προτεινόμενων διαμορφώσεων που θα εξάγει ο Ενορχηστρωτής
- **max\_parallel\_configurations:** Ο μέγιστος αριθμός των Ροών Διαμόρφωσης που θα τρέχουν παράλληλα.
- **tuner:** Ένα αντικείμενο spec ενός πειράματος Katib το οποίο θα χρησιμοποιηθεί για περαιτέρω βελτιστοποίηση παραμέτρων του καλύτερου μοντέλου.

Όπως αναφέραμε στην αρχή αυτού του κεφαλαίου, η συνάρτηση διεπαφής `run_automl()` επιστρέφει ένα αντικείμενο ***AutoMLExperiment*** (4.4.2). Αυτό το αντικείμενο, και ουσιαστικά οι μέθοδοι του, θα επιτρέψουν στον χρήστη να παρακολουθεί την κατάσταση του πειράματος. Θα αναπτύξουμε περισσότερο την παρακολούθηση της κατάστασης του πειράματος στην ενότητα 4.4, αφού πρώτα έχουμε αναλύσει την λειτουργικότητα των διοχετεύσεων του **Ενορχηστρωτή** (4.2) και των **Ροών Διαμόρφωσης** (4.3).

Παρακάτω παρατίθεται μια **αριθμημένη λίστα με βήματα** που περιγράφει πως η συνάρτηση διεπαφής `run_automl()` καταφέρνει να πυροδοτήσει ολόκληρη την διαδικασία AutoML:

1. **Χρησιμοποιεί τον αποθηκευτικό μηχανισμό του Kale για να αποθηκεύσει τις παραμέτρους εισόδου της στον λογικό δίσκο του Pod και ύστερα τραβάει ένα στιγμιότυπο Rok του δίσκου** έτσι ώστε τα βήματα του Ενορχηστρωτή και των Ροών Διαμόρφωσης μπορούν να προσαρτήσουν τον κλωνοποιημένο δίσκο για να βρουν και να φορτώσουν τις παραμέτρους.
2. **Κατασκευάζει ένα αντικείμενο *kale.PipelineConfig* object** με όλες τις βασικές περιγραφικές πληροφορίες (π.χ: το όνομα) του Ενορχηστρωτή.
3. **Κατασκευάζει ένα αντικείμενο *kale.processors.PythonProcessor*** (2.5.5) με μια συνάρτηση Python γραμμένη σε γλώσσα συγκεκριμένου τομέα του Kale (2.5.4). Αυτή η συνάρτηση Python ονομάζεται `automl_orchestrate` και ουσιαστικά περιγράφει ολόκληρη την λειτουργικότητα μηχανικής μάθησης του Ενορχηστρωτή. Εκθέτουμε την αρχιτεκτονική της συνάρτησης `automl_orchestrate` στην υπο-ενότητα 4.2.1. Το αντικείμενο *PythonProcessor* (2.5.5) αξιολογεί αυτή την συνάρτηση διοχέτευσης και επιστρέφει ένα αντικείμενο *Pipeline* (2.5.3).
4. **Μεταγλωττίζει το αντικείμενο Pipeline σε ένα Argo Workflow και το στέλνει στον Κυβερνήτη.** Αυτό συμβαίνει χρησιμοποιώντας το αντικείμενο *kale.Compiler* και την `compile_and_run` μέθοδο του (2.5.6).
5. **Επιστρέφει ένα αντικείμενο *AutoMLExperiment*, αρχικοποιημένο με το ID του προσφάτως δημιουργημένου Ενορχηστρωτή.** Αυτό το αντικείμενο επιτρέπει στον

χρήστη να παρακολουθεί την κατάσταση του Ενορχηστρωτή και των Ροών Διαμόρφωσης και επίσης επιτρέπει την επίβλεψη των αποτελεσμάτων μετρικής των Ροών Διαμόρφωσης, μόλις αυτά γίνουν διαθέσιμα.

### 4.1.1 Το Αντικείμενο Dataset

Προκειμένου να απλοποιήσουμε το αποτύπωμα (signature) της συνάρτησης `run_automl()`, αποφασίσαμε να αναπαραστήσουμε το σύνολο των δεδομένων μηχανικής μάθησης του χρήστη και όλα τα επι-μέρους στοιχεία του σας μία αφηρημένη οντότητα. Αυτή η οντότητα είναι ένα αντικείμενο Dataset, και παρακάτω φαίνονται τα χαρακτηριστικά του:

- **name**
- **features**
- **targets**
- **features\_test**
- **targets\_test**

## 4.2 Ο Ενορχηστρωτής AutoML

Σε αυτή την ενότητα θα περιγράψουμε την διοχέτευση που δημιουργεί και ελέγχει ολόκληρο το πείραμα AutoML. Αυτή η διοχέτευση ονομάζεται **Ενορχηστρωτής AutoML**, και ουσιαστικά αποτελείται από έξι βήματα τα οποία τρέχουν κώδικα του Kale. Παρακάτω παρατίθεται μια περιγραφή υψηλού επιπέδου του μηχανισμού που υλοποιεί το κάθε βήμα:

1. **get-metalearning-configurations:** Χρησιμοποιώντας τον πυρήνα του auto-sklearn, αυτό το βήμα εξάγει ένα σύνολο από μετα-χαρακτηριστικά από το σύνολο δεδομένων εισόδου και παράγει μια λίστα από προτεινόμενες διαμορφώσεις μετα-μάθησης.
2. **run-metalearning-configurations:** Αυτό το βήμα παίρνει τις διαμορφώσεις του βήματος 1 και δημιουργεί μία νέα Ροή Διαμόρφωσης για κάθε μια από αυτές.
3. **monitor-kfp-runs:** Αυτό το βήμα περιμένει τις Ροές Διαμόρφωσης που δημιουργήθηκαν στο βήμα 2 να ολοκληρωθούν.
4. **get-best-configuration:** Αυτό το βήμα συγκεντρώνει τα σκορ μετρικής από κάθε Ροή Διαμόρφωσης και επιλέγει την Ροή Ρύθμισης που είχε την καλύτερη επίδοση.
5. **run-katib-experiment:** Σε αυτό το βήμα, το Kale παίρνει τις διαμορφώσεις μετα-μάθησης που αντιστοιχούν στο καλύτερη Ροή Διαμόρφωσης του βήματος 4, και δημιουργεί ένα πείραμα Katib που θα υλοποιεί διαμόρφωση υπερ-παραμέτρων για το μοντέλο.
6. **monitor-katib-experiment:** Αυτό το βήμα περιμένει το πείραμα Katib του προηγούμενου βήματος να ολοκληρωθεί.

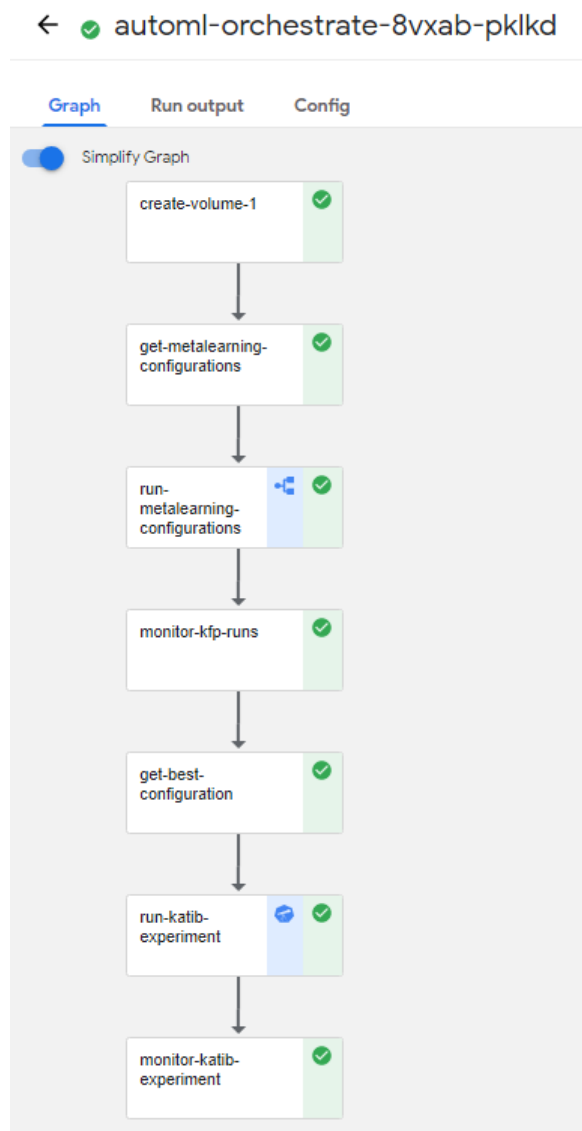


Figure 4.1: Παράδειγμα μίας διοχέτευσης Ενορχηστρωτή AutoML από την διεπαφή χρήστη του KFP

#### 4.2.1 Η Συνάρτηση Διοχέτευσης `automl_orchestrate`

Η συνάρτηση `automl_orchestrate` είναι μια συνάρτηση Python, γραμμένη στην γλώσσα συγκεκριμένου τομέα του Kale (2.5.4), η οποία ουσιαστικά περιγράφει την δομή του Ενορχηστρωτή AutoML που δημιουργεί η συνάρτηση `run_automl`. Εφόσον είναι γραμμένη στην γλώσσα συγκεκριμένου τομέα του Kale, χρησιμοποιεί `Step` αντικείμενα (2.5.2) για να αναπαραστήσει βήματα σε μία διοχέτευση. Παρακάτω παραθέτουμε τον κώδικα της συνάρτησης:

Listing 4.2: Η συνάρτηση διοχέτευσης `automl_orchestrate`

```

1 def automl_orchestrate(hp_tune="false"):
2     """Auto ML pipeline."""
3     (configurations,
4      kale_dataset_id) = GetMetaLearningConfigurations()(
5         ml_assets_marshal_path("dataset.dill.pkl"),

```

```

6     ml_assets_marshall_path("task.dillpkl"),
7     ml_assets_marshall_path("metric.joblib"),
8     ml_assets_marshall_path("number_of_configurations.dillpkl"))
9
10    run_ids = RunMetaLearningConfigurations()(
11        configurations,
12        ml_assets_marshall_path("max_parallel_configurations.dillpkl"),
13        kale_dataset_id)
14    run_ids = MonitorKFPRuns()(run_ids)
15    best_configuration = GetBestConfiguration()(
16        run_ids,
17        configurations,
18        ml_assets_marshall_path("metric.joblib"))
19    if hp_tune == "true":
20        katib_experiment_name = RunKatibExperiment()(
21            ml_assets_marshall_path("tuner.dillpkl"),
22            best_configuration,
23            ml_assets_marshall_path("metric.joblib"),
24            kale_dataset_id)
25        MonitorKatibExperiment()(katib_experiment_name)

```

Παρατηρήστε ότι μερικές από τις παραμέτρους εισόδου των βημάτων είναι **μονοπάτια σε σειριοποιημένα αντικείμενα** που αντιστοιχούν σε αποθηκευμένες παραμέτρους εισόδου της συνάρτησης `run_automl` (4.1). Όπως είναι φανερό από τον προηγούμενο κώδικα, η διοχέτευση του Εννορησιρωτή AutoML την οποία περιγράφει η `automl_orchestrate()`, αποτελείται από έξι Kale Steps:

1. **GetMetaLearningConfigurations**
2. **RunMetaLearningConfigurations**
3. **MonitorKFPRuns**
4. **GetBestConfiguration**
5. **RunKatibExperiment**
6. **MonitorKatibExperiment**

Κάθε ένα από αυτά είναι ένα κανονικό βήμα διοχέτευσης Kale που υλοποιεί ένα συγκεκριμένο τμήμα του Εννορησιρωτή AutoML. Στις επόμενες ενότητες, θα εκθέσουμε την λειτουργικότητα καθενός από αυτά τα βήματα.

#### 4.2.2 Το Βήμα *GetMetaLearningConfigurations*

Σε αυτή την υποενότητα παρουσιάζουμε την λειτουργικότητα που υλοποιεί το βήμα *GetMetaLearningConfigurations*.

Listing 4.3: Η μέθοδος `do_run()` του βήματος *GetMetaLearningConfigurations*

```

1 class GetMetaLearningConfigurations(Step):
2     """Produce ML suggestions from a dataset using AutoSKLearn MetaLearning.
3

```



```

4     Ins:
5         dataset (Dataset):
6         task:
7         metric (Callable):
8         number_of_configurations (int)
9
10    Outs:
11        configurations (List[Configuration])
12        kale_dataset_id (int)
13
14    MLMD Inputs:
15        kale.Dataset
16
17    MLMD Outputs:
18        kale.AutoMLConfiguration (##'number_of_configurations')
19    """
20    name = "get-metalearning-configurations"
21    outs = Param.odict(["configurations", "kale_dataset_id"], step_name=name)
22
23    def do_run(self, dataset, task, metric, number_of_configurations):
24        """Implementations of GetMetaLearningConfigurations."""
25        from kale.ml import metalearning
26
27        dataset_artifact = self._submit_and_link_dataset_artifact(dataset)
28        configurations = metalearning.compute_configs(dataset, task, metric,
29                                                    number_of_configurations)
30        for idx, configuration in enumerate(configurations):
31            self._submit_configuration_artifact(configuration, idx)
32        return configurations, dataset_artifact.id

```

#### 4.2.2.1 Η Συνάρτηση *compute\_configs*

Η διαδικασία υπολογισμού των διαμορφώσεων μηχανικής μάθησης υλοποιείται ουσιαστικά από μια συνάρτηση του δομοστοιχείου *kale.ml.metalearning*, που ονομάζεται: ***compute\_configs***. Παρακάτω παραθέτουμε τον κώδικα που τρέχει η συνάρτηση *compute\_configs*:

Listing 4.4: Η συνάρτηση *compute\_configs* που παράγει μια λίστα με προτεινόμενες διαμορφώσεις διοχέτευσης μηχανικής μάθησης

```

1    def compute_configs(dataset: Dataset,
2                        task: MLTask,
3                        metric: Callable,
4                        number_of_configurations: int) -> List[Configuration]:
5        """Use the AutoSKLearn MetaLearning system to produce ML configurations.
6
7        The AutoSKLearn MetaLearning system is based on prior knowledge on how
8        certain Machine Learning models perform on a set of known datasets.
9        AutoSKLearn can use this prior knowledge to suggest some Machine Learning

```

```

10     configurations that are supposed to perform well on a new, previously
11     unseen, dataset."""
12     log.info("Getting suggested configurations...")
13
14     task = mltask_to_string(task)
15
16     validated_dataset, feature_types = _validate_dataset(dataset, task)
17     task_type = extract_task_type(y=validated_dataset.targets, task=task)
18
19     metafeatures = calculate_all_metafeatures(x=validated_dataset.features,
20                                             y=validated_dataset.targets,
21                                             dataset_name=dataset.name,
22                                             task_type=task_type,
23                                             feature_types=feature_types)
24
25     # XYDataManager does some validation to the dataset and the list of feature
26     # types. It also finds if the dataset is sparse or not - useful for
27     # detecting the metadata directory.
28     datamanager = XYDataManager(X=validated_dataset.features,
29                                y=validated_dataset.targets,
30                                X_test=validated_dataset.features_test,
31                                y_test=validated_dataset.targets_test,
32                                task=task_type,
33                                feat_type=feature_types,
34                                dataset_name=dataset.name)
35     is_sparse = datamanager.info["is_sparse"]
36
37     metadata_directory = find_metadata_dir(task_type, metric, is_sparse)
38     config_space = get_configuration_space(datamanager.info)
39     # The MetaBase object is a container for metafeatures, configurations
40     # and their respective scores.
41     meta_base = MetaBase(config_space, metadata_directory)
42     meta_base.add_dataset(dataset.name, metafeatures)
43     configurations = suggest_via_metalearning(
44         meta_base=meta_base, dataset_name=dataset.name,
45         metric=metric, task=task_type, sparse=is_sparse,
46         num_initial_configurations=number_of_configurations)
47     return configurations

```

#### 4.2.2.2 Η Συνάρτηση `_validate_dataset`

Η συνάρτηση `_validate_dataset`, όπως είναι εμφανές από το όνομα της, ελέγχει το σύνολο δεδομένων μηχανικής μάθησης που παρέχει ο χρήστης. Ανήκει στο δομοστοιχείο `kale.ml.metalearning` του Kale και ουσιαστικά χρησιμοποιεί την κλάση `InputValidator` (sub-section 3.5.1) από το δομοστοιχείο `autosklearn.data.validation` του auto-sklearn για να ελέγξει το σύνολο δεδομένων εισόδου. Ακολουθώντας, παρουσιάζουμε τον κώδικα της συνάρτησης `_validate_dataset`.

Listing 4.5: Η συνάρτηση `_validate_dataset` του δομοστοιχείου `kale.ml.metalearning`

```

1 def _validate_dataset(dataset: Dataset,
2                       task: str = "classification") -> (
3                       Tuple[Dataset, List[str]]):
4     """Validate and process the input features and targets.
5
6     Use AutoSKLearn "InputValidator" to check if the input dataset
7     is valid (e.g: the number of samples matches the number of targets).
8     Also, auto-sklearn does some "polishing" transformations to the dataset.
9
10    During the validation, "InputValidator" also determines the feature
11    type for all the input features. A feature type can either be "numerical"
12    or "categorical".
13
14    Args:
15        dataset (Dataset): A Dataset class object that contains the input
16        dataset for the ML task.
17        task (str): The type of the ML task (classification | regression).
18
19    Returns:
20        Dataset, List(str): The validated dataset and a list of feature types
21        for all features (either "numerical" or "categorical").
22    """
23    is_classification = (task == "classification")
24    input_validator = InputValidator()
25    x, y = input_validator.validate(X=dataset.features, y=dataset.targets,
26                                  is_classification=is_classification)
27    x_test, y_test = input_validator.validate(
28        X=dataset.features_test, y=dataset.targets_test,
29        is_classification=is_classification)
30
31    validated_dataset = copy.deepcopy(dataset)
32    validated_dataset.features = x
33    validated_dataset.targets = y
34    validated_dataset.features_test = x_test
35    validated_dataset.targets_test = y_test
36    return validated_dataset, input_validator.feature_types

```

#### 4.2.2.3 Η Συνάρτηση `find_metadata_dir`

Η συνάρτηση `find_metadata_dir` του δομοστοιχείου `kale.ml.utils` βρίσκει το μονοπάτι στον κατάλογο μετα-δεδομένων που αντιστοιχεί σε μια δοσμένη εργασία μηχανικής μάθησης. Παρακάτω φαίνεται ο κώδικας της συνάρτησης:

Listing 4.6: Η συνάρτηση `find_metadata_dir` που βρίσκει το σωστό μονοπάτι στον κατάλογο μετα-δεδομένων για μία δοσμένη εργασία μηχανικής μάθησης

```

1 def find_metadata_dir(task_type: int, metric: Callable, is_sparse: int):

```

```

2     """Find the directory where auto-sklearn stores its meta-knowledge.
3
4     Note:
5         The directory structure follows the 'Algorithm Selection Library'
6         (ASLib) format. See https://www.automl.org/automated-algorithm-design/algorithm-
7         selection/aslib/ # noqa: 501
8
9     Args:
10        task_type (int): The type of the ML task.
11        metric (callable): The metric that is used for the ML task.
12        is_sparse (int): Whether the dataset is sparse or not.
13
14    Returns:
15        str: path to the auto-sklearn metadata directory.
16
17    Raises:
18        RuntimeError: When the auto-sklearn metadata directory cannot be found.
19    """
20    log.info("Finding metadata dir...")
21    metalearning_directory = os.path.dirname(autosklearn.metalearning.__file__)
22    # The auto-sklearn metadata directory doesn't provide metadata for
23    # multi-label classification. auto-sklearn reverts to using binary
24    # classification as well, so we copy this behaviour.
25    if task_type == constants.MULTILABEL_CLASSIFICATION:
26        meta_task = constants.BINARY_CLASSIFICATION
27    else:
28        meta_task = task_type
29    metalearning_files_dir = "%s_%s_%s" % (
30        metric, constants.TASK_TYPES_TO_STRING[meta_task],
31        "sparse" if is_sparse else "dense")
32    metadata_directory = os.path.join(
33        metalearning_directory, "files", metalearning_files_dir)
34    if not os.path.exists(metadata_directory):
35        raise RuntimeError("Metadata directory %s does not exist."
36                           % metadata_directory)
37    log.info("Metadata directory: %s", metadata_directory)
38    return metadata_directory

```

#### 4.2.2.4 Η Μέθοδος `_submit_configuration_artifact`

Προκειμένου να διατηρήσουμε μια γενεαλογία ολόκληρου του πειράματος AutoML, αναγκάζουμε τα βήματα διοχετεύσεων να δημιουργούν και να υποβάλλουν Artifacts στην βάση δεδομένων MLMD. Η μέθοδος `_submit_configuration_artifact` δημιουργεί ένα *AutoML-Configuration Artifact* (subsubsection 4.4.1.2) που αντιστοιχεί σε μια δοσμένη διαμόρφωση διοχέτευσης.

Listing 4.7: Η μέθοδος `_submit_configuration_artifact` του `GetMetaLearningConfigurations` που δημιουργεί και υποβάλλει ένα `AutoMLConfiguration` Artifact για μια δοσμένη διαμόρφωση διοχέτευσης

```

1 def _submit_configuration_artifact(self, configuration, idx):
2     from kale.ml import utils
3     from kale.common.artifacts import AutoMLConfiguration
4
5     mlmd = mlmdutils.get_mlmd_instance()
6     config_summary = utils.get_configuration_summary(configuration)
7     config = AutoMLConfiguration(
8         config_summary=config_summary,
9         run_id=kfputils.format_kfp_run_id_uri(mlmd.run_uuid),
10        estimator_name="Configuration %s: %s"
11            % (idx + 1, config_summary["name"]))
12    config.assign_list_index(idx + 1)
13    config_artifact = config.submit_artifact()
14    mlmd.link_artifact_as_output(config_artifact.id)

```

Για να παράξει την παράμετρο `config_summary`, το βήμα αυτό χρησιμοποιεί την συνάρτηση `get_configuration_summary` του δομοστοιχείου `ml.utils`:

Listing 4.8: Η συνάρτηση `get_configuration_summary` που δημιουργεί ένα λεξικό-περίληψη μιας δοσμένης διαμόρφωσης διοχέτευσης

```

1 def get_configuration_summary(configuration: Configuration) -> Dict[str, Any]:
2     """Return an opinionated summary of the input configuration.
3
4     The output of this function can be used to pretty-print a configuration,
5     with just the right information, or to upload the configuration to the
6     artifact store.
7
8     Args:
9         configuration: A suggested configuration extracted by auto-sklearn.
10
11    Returns:
12        dict: A dictionary that describes the learner (classifier, or
13            regressor) of the configuration, with the following fields:
14
15            * 'name': The name of the model
16            * 'parameters': A dictionary of hyperparameters.
17
18    Raises:
19        ValueError: If cannot find a supported learner type. Supported
20            learner types are 'classifier:__choice__' and
21            'regressor:__choice__'.
22    """
23    if configuration.get("classifier:__choice__"):
24        name = _get_classifier_name(configuration)
25        params = _get_params(configuration, "classifier")

```

```

26 elif configuration.get("regressor:__choice__"):
27     name = _get_regressor_name(configuration)
28     params = _get_params(configuration, "regressor")
29 else:
30     raise ValueError("Could not find a model in the input configuration.")
31
32 return {"name": name, "parameters": params}

```

### 4.2.3 Το Βήμα *RunMetaLearningConfigurations*

Σε αυτήν την υποενότητα θα παρουσιάσουμε και θα εξηγήσουμε τον μηχανισμό που υλοποιεί το βήμα *RunMetaLearningConfigurations*. Συνοπτικά, αυτό το βήμα λαμβάνει ως είσοδο την λίστα των διαμορφώσεων διοχέτευσης που δημιούργησε το προηγούμενο βήμα, και για κάθε μια τους, δημιουργεί μία νέα Ροή Διαμόρφωσης που υλοποιεί την διοχέτευση μηχανικής μάθησης που η αντίστοιχη διαμόρφωση διοχέτευσης περιγράφει. Ας δούμε τον κώδικα που εκτελείται μέσα στο βήμα αυτό, ξεκινώντας από τον ορισμό της κλάσης του και την μέθοδο *do\_run*:

Listing 4.9: Ο ορισμός κλάσης και η μέθοδος *do\_run()* του βήματος *RunMetaLearningConfigurations*

```

1 class RunMetaLearningConfigurations(Step):
2     """Run MetaLearning suggestions as KFP pipelines.
3
4     Ins:
5         configurations (List[Configuration])
6         max_parallel_configurations (int)
7         kale_dataset_id (int)
8
9     Outs:
10        run_ids (List[str])
11        """
12    name = "run-metalearning-configurations"
13    outs = Param.odict(["run_ids"], step_name=name)
14    actions = ["RunKFPpipelines"]
15
16    def do_run(self, configurations, max_parallel_configurations,
17              kale_dataset_id):
18        """Implementation of RunMetaLearningConfigurations."""
19        from time import sleep
20        from kale import marshal
21        from kale.common import mlmdutils
22        from kale.ml.utils import ML_ASSETS_DIR
23        from kale.common.artifacts import AutoMLConfiguration
24
25        self.vars["run_ids"] = []
26        marshal.set_data_dir(ML_ASSETS_DIR)
27

```

```

28     mlmd = mlmdutils.get_mlmd_instance()
29     # Get all Artifacts that are attributed to the Context of the AutoML
30     # Orchestrator. The SKLearnTransformer step of each new Configuration
31     # Run should link the corresponding AutoMLConfiguration Artifact as its
32     # input. So, we should pass the corresponding AutoMLConfiguration
33     # Artifact ID to each Configuration Run.
34     kale_config_artifacts = mlmdutils.get_artifacts_by_context_and_type(
35         context_id=mlmd.run_context.id,
36         type_name=AutoMLConfiguration.artifact_type_name,
37         sorted=True)
38     if len(kale_config_artifacts) != len(configurations):
39         raise RuntimeError("Founds %d MLMD configuration artifacts but"
40                             " %d configuration were provided as input."
41                             % (len(kale_config_artifacts),
42                                len(configurations)))
43
44     # Start all configurations with a reconciliation loop to avoid having
45     # more than max_parallel_configurations running concurrently
46     while configurations:
47         if self._running_ids() >= max_parallel_configurations:
48             log.info("Cannot start a new configuration. Max parallel"
49                     " configurations cap is set to %d. Waiting for a"
50                     " configuration to complete...",
51                     max_parallel_configurations)
52             sleep(10)
53             continue
54
55         configuration = configurations.pop(0)
56         index = len(self.vars["run_ids"]) + 1
57
58         log.info("Saving configuration n. %d", index)
59         marshal.save(configuration, "configuration")
60
61         automl_config_artifact_id = kale_config_artifacts[index - 1].id
62         run_id = self._run_pipeline(index, {
63             "kale_dataset_id": str(kale_dataset_id),
64             "kale_config_id": str(automl_config_artifact_id)})
65         self.vars["run_ids"].append(run_id)
66         self._patch_context(run_id, index)
67
68     return self.vars["run_ids"]

```

#### 4.2.3.1 Η Μέθοδος `_run_pipeline`

Όταν ο αριθμός των εκτελούμενων Ροών Διαμόρφωσης είναι μικρότερος από την τιμή της παραμέτρου `max_parallel_configurations` (παράμετρος εισόδου της συνάρτησης `run_automl`), το βήμα `RunMetaLearningConfigurations` επιλέγει μια διαμόρφωση διοχέτευσης και δημιουργεί μία νέα Ροή Ρύθμισης που να την υλοποιεί. Η δημιουργία της Ροής Ρύθμισης

υλοποιείται από την μέθοδο `_run_pipeline` της κλάσης `RunMetaLearningConfigurations`.

Listing 4.10: Η μέθοδος `_run_pipeline` της κλάσης `RunMetaLearningConfigurations`

```

1  def _run_pipeline(self, index, params: Dict = {}):
2      from kale.types import Param
3      from kale import PipelineConfig, Compiler
4      from kale.processors import PythonProcessor
5      from kale.ml.pipelines import sklearn_train_predict
6
7      volumes = rokutils.interactive_snapshot_and_get_volumes()
8      pipeline_config = PipelineConfig(
9          pipeline_name="sklearn-configuration-%d" % index,
10         experiment_name=kfputils.get_experiment_from_run_id(
11             kfputils.detect_run_uuid()).name,
12         marshal_path=self.marshal_path,
13         volumes=volumes)
14         pipeline_params = {k: Param(type(v).__name__, v)
15                             for k, v in params.items()}
16
17         log.newline()
18         log.info("Creating pipeline for configuration n. %d", index)
19         processor = PythonProcessor(
20             sklearn_train_predict.sklearn_train_predict, pipeline_config)
21         processor.pipeline.default_pipeline_parameters.update(pipeline_params)
22         pipeline = processor.run()
23         log.info("Running pipeline for configuration n. %d", index)
24         run = Compiler(pipeline).compile_and_run()
25         log.info("Successfully started run %s" % run.id)
26         log.newline()
27         return run.id

```

Όπως περιγράψαμε παραπάνω, η μέθοδος `_run_pipeline` ουσιαστικά δημιουργεί πραγματικές διοχετεύσεις Kubeflow, οι οποίες ονομάζονται Ροές Ρύθμισης. Για τον σκοπό αυτό η μέθοδος ουσιαστικά χρησιμοποιεί το back-end του Kale (section 2.5).

#### 4.2.4 Το Βήμα `MonitorKFPRuns`

Σε αυτή την υποενότητα θα εκθέσουμε τον μηχανισμό που υλοποιεί το βήμα `MonitorKFPRuns`. Συνοπτικά, αυτό το βήμα παρακολουθεί την κατάσταση όλων των Ροών Διαμόρφωσης και μόλις ολοκληρωθούν όλες τους, επιτυχημένα ή όχι, το βήμα ολοκληρώνει την εκτέλεση του. Ας παρουσιάσουμε τον κώδικα που τρέχει μέσα σε αυτό το βήμα, και πιο συγκεκριμένα τον ορισμό κλάσης του και την μέθοδο `do_run`:

Listing 4.11: Ο ορισμός κλάσης και η μέθοδος `do_run()` του βήματος `MonitorKFPRuns`

```

1  class MonitorKFPRuns(Step):
2      """Wait for KFP pipelines to complete.
3
4      Ins:
5          run_ids

```



```

6
7     Outs:
8         run_ids
9     """
10    name = "monitor-kfp-runs"
11    outs = Param.odict({"run_ids"}, step_name=name)
12
13    def do_run(self, run_ids):
14        """Implementation of MonitorKFPRuns."""
15        from time import sleep
16
17        log.info("Monitoring runs: %s", run_ids)
18        statuses = {run_id: "Pending" for run_id in run_ids}
19
20        # Add custom properties linking the MLMD Execution with the runs to
21        # monitor
22        log.info("Patching MLMD Execution custom properties with the"
23                " configuration run IDs...")
24        mlmd = mlmdutils.get_mlmd_instance()
25        custom_props = {"configuration_run_%d" % idx:
26                        kfputils.format_kfp_run_id_uri(run_ids[idx])
27                        for idx in range(len(run_ids))}
28        mlmdutils.patch_execution_custom_properties(mlmd.execution.id,
29                                                    custom_props)
30        log.info("Successfully patched MLMD Execution")
31
32        while any(map(lambda state: state not in kfputils.KFP_RUN_FINAL_STATES,
33                      statuses.values())):
34            log.newline()
35            log.info("Updating pipelines statuses...")
36            for run_id in statuses.keys():
37                if statuses[run_id] not in kfputils.KFP_RUN_FINAL_STATES:
38                    statuses[run_id] = kfputils.get_run(run_id).run.status
39                    log.info("Run '%s': %s", run_id, statuses[run_id])
40            sleep(5)
41
42        log.info("All done!")
43        return run_ids

```

Όπως μπορούμε να δούμε στον παραπάνω κώδικα, το αποτύπωμα της μεθόδου *do\_run* του βήματος είναι αρκετά απλή. Λαμβάνει μία λίστα από IDs Ροών Διαμόρφωσης ως είσοδο, και την επιστρέφει μόλις τελειώσει την εκτέλεση του, ώστε να την λάβει το επόμενο βήμα. Η λίστα με τα IDs των Ροών Διαμόρφωσης θα χρησιμοποιηθεί για να ερωτηθεί ο KFP server για την κατάσταση των Ροών Διαμόρφωσης.

### 4.2.5 Το Βήμα *GetBestConfiguration*

Σε αυτή την υποενότητα θα παρουσιάσουμε και θα εξηγήσουμε τον μηχανισμό που υλοποιεί το βήμα *GetBestConfiguration*. Συνοπτικά, αυτό το βήμα παίρνει ως είσοδο τη λίστα με τα IDs των Ροών Διαμόρφωσης, συλλέγει τα σκορ μετρικής των επιτυχημένων Ροών Διαμόρφωσης, βρίσκει την Ροή Διαμόρφωσης με το καλύτερο σκορ και επιστρέφει το αντίστοιχο αντικείμενο διαμόρφωσης μηχανικής μάθησης.

Listing 4.12: Ο ορισμός κλάσης και η μέθοδος *do\_run()* του βήματος *GetBestConfiguration*

```

1  class GetBestConfiguration(Step):
2      """Get the best-performing MetaLearning configuration.
3
4      Ins:
5          run_ids
6          configurations
7          metric
8
9      Outs:
10         best_configuration
11     """
12     name = "get-best-configuration"
13     outs = Param.odict(["best_configuration"], step_name=name)
14
15     def do_run(self, run_ids, configurations, metric):
16         """Implementation of GetBestConfiguration."""
17         metrics = dict()
18         for run_id in run_ids:
19             log.info("Collecting metrics for run: %s", run_id)
20             metrics[run_id] = kfputils.get_kfp_run_metrics(run_id)
21         final_metrics = metrics.copy()
22
23         log.newline()
24         log.info("Collected metrics: \n")
25         for run_id, _metrics in metrics.items():
26             log.info("  Run %s:", run_id)
27             if not _metrics.values():
28                 log.info("    No metrics found.")
29                 del final_metrics[run_id]
30             for name, value in _metrics.items():
31                 log.info("    %s: %s", name, value)
32         log.info("Using metric '%s' as target metric.", metric.name)
33
34         # Get best metric, excluding empty metrics dictionaries
35         opt = max if metric._sign == 1 else min # see arrikto/dev#1128
36         best_run_uid, best_metric = opt(final_metrics.items(),
37                                       key=lambda x: x[1][metric.name])
38         log.info("Best run id: %s", best_run_uid)
39
40         log.info("Patching MLMD Execution and Context custom properties with"

```

```

41         " the best configuration run ID...")
42     custom_prop = {"best_configuration_run":
43                   kfputils.format_kfp_run_id_uri(best_run_uid)}
44     mlmd = mlmdutils.get_mlmd_instance()
45     mlmdutils.patch_execution_custom_properties(mlmd.execution.id,
46                                                custom_prop)
47     log.info("Successfully patched MLMD Execution")
48     mlmdutils.patch_context_custom_properties(mlmd.run_context.id,
49                                              custom_prop)
50     log.info("Successfully patched MLMD Context")
51
52     best_configuration = configurations[run_ids.index(best_run_uid)]
53     return best_configuration

```

### 4.2.6 Το Βήμα *RunKatibExperiment*

Σε αυτήν την υποενότητα, θα παρουσιάσουμε τον μηχανισμό που υλοποιεί το βήμα *RunKatibExperiment*. Το βήμα αυτό παίρνει την διαμόρφωση διοχέτευσης με το μεγαλύτερο σκορ μετρικής από το προηγούμενο βήμα, και δημιουργεί ένα πείραμα (2.4.4) ώστε να βελτιώσει περαιτέρω την διοχέτευση αυτή.

Αυτό και το επόμενο βήμα (*MonitorKatibExperiment*) εκτελούνται **μόνο στην περίπτωση που ο χρήστης έχει ορίσει ένα αντικείμενο *tuner*** (*katib.ExperimentSpec* object) ως παράμετρο εισόδου στην συνάρτηση *run\_automl* (section 4.1). Αν δεν έχει περαστεί κάποια παράμετρος εισόδου *tuner* στην συνάρτηση *run\_automl*, τότε το *GetBestConfiguration* (subsection 4.2.5) είναι το τελευταίο βήμα του Ενορχηστρωτή AutoML. Παρακάτω παρουσιάζουμε τον ορισμό κλάσης και τις κύριες μεθόδους του βήματος:

Listing 4.13: Ο ορισμός κλάσης και οι κύριες μέθοδοι του βήματος *RunKatibExperiment*

```

1 class RunKatibExperiment(Step):
2     """Run a Katib experiment.
3
4     Ins:
5         tuner (katib.V1beta1ExperimentSpec): Experiment spec
6         best_configuration (ConfigSpace.Configuration):
7         metric (autosklearn.metric): An (Auto)SKLearn metric
8         kale_dataset_id (int): The MLMD artifact ID
9
10    Outs:
11        katib_experiment_name (str): Katib experiment name
12    """
13    name = "run-katib-experiment"
14    outs = Param.odict({"katib_experiment_name"}, step_name=name)
15    actions = ["KatibExperiment"]
16
17    def _get_hyperparams(self, configuration):
18        return {
19            hp_name: configuration[hp_name]

```

```

20     for hp_name in configuration.keys()
21         if (any(map(lambda alg_type: hp_name.startswith(alg_type),
22                     ["classifier", "regressor"])))
23             and not hp_name.endswith("__choice__"))
24
25     def _generate_hyperparam_conf(self, key, value, conf_space):
26         conf_space = copy.deepcopy(conf_space)
27         hp = conf_space.get_hyperparameter(key)
28         if hasattr(hp, "upper") or hasattr(hp, "choices"):
29             hp.default_value = value
30             return hp
31         return None
32
33     def _get_hyperparams_confs(self, configuration):
34         configuration = copy.deepcopy(configuration)
35         hyperparams_conf = []
36         for key, value in self._get_hyperparams(configuration).items():
37             hp = self._generate_hyperparam_conf(
38                 key, value, configuration.configuration_space)
39             if hp:
40                 hyperparams_conf.append(hp)
41         return hyperparams_conf
42
43     def _get_param(self, conf):
44         from kubeflow import katib
45         if hasattr(conf, "choices"):
46             return katib.V1beta1ParameterSpec(
47                 feasible_space=katib.V1beta1FeasibleSpace(
48                     list=list(conf.choices)),
49                 name=self._conf_name(conf),
50                 parameter_type="categorical")
51         else: # for now assume just float ranges
52             return katib.V1beta1ParameterSpec(
53                 feasible_space=katib.V1beta1FeasibleSpace(
54                     max=str(float(conf.upper)),
55                     min=str(float(conf.lower)),
56                     # heuristic just for test purposes
57                     step=str((float(conf.upper) - float(conf.lower)) / 10)),
58                 name=self._conf_name(conf),
59                 parameter_type="double")
60
61     def do_run(self, tuner, best_configuration, metric, kale_dataset_id):
62         """Implementation of RunKatibExperiment."""
63         from kubeflow import katib
64         from kale.types import Param
65         from kale.processors import PythonProcessor
66         from kale import marshal, PipelineConfig, Compiler
67         from kale.ml.pipelines import sklearn_train_predict

```

```

68
69     hyperparam_confs = self._get_hyperparams_confs(best_configuration)
70
71     # Marshal the configuration, used by the train-predict pipeline
72     marshal.save(best_configuration, "configuration")
73
74     # Configure the HP tuning settings
75     tuner.algorithm = katib.V1beta1AlgorithmSpec(algorithm_name="grid")
76     tuner.objective = katib.V1beta1ObjectiveSpec(
77         objective_metric_name=metric.name,
78         type=tuner.objective.type or "maximize")
79     tuner.parameters = [self._get_param(conf) for conf in hyperparam_confs]
80
81     volumes = rokutils.interactive_snapshot_and_get_volumes()
82     log.info("Creating parametrized pipeline...")
83     pipeline_config = PipelineConfig(
84         pipeline_name="katib-trial-sklearn-configuration",
85         experiment_name=kfputils.get_experiment_from_run_id(
86             kfputils.detect_run_uuid()).name,
87         marshal_path=self.marshal_path,
88         volumes=volumes,
89         katib_metadata=tuner,
90         katib_run=True)
91     processor = PythonProcessor(
92         sklearn_train_predict.sklearn_train_predict, pipeline_config)
93     pipeline = processor.run()
94
95     for conf in hyperparam_confs:
96         pipeline.default_pipeline_parameters[
97             self._conf_name(conf)] = Param(name=self._conf_name(conf),
98                                           param_type="str")
99
100     def _patch_step_cli(step: BaseStep):
101         del step.ins["masked_inputs"]
102         for conf in hyperparam_confs:
103             name = ".msk.%s" % self._conf_name(conf)
104             step.ins[name] = Param(name=name)
105     _patch_step_cli(pipeline.get_step("run-sklearn-transformer"))
106     _patch_step_cli(pipeline.get_step("train-sklearn-estimator"))
107
108     log.info("Running Katib experiment...")
109     experiment = Compiler(pipeline).compile_and_run()
110     log.info(experiment)
111
112     self._patch_experiment_uri(experiment["metadata"]["name"],
113                              experiment["metadata"]["namespace"])
114     return experiment["metadata"]["name"]

```

### 4.2.7 Το Βήμα *MonitorKatibExperiment*

Αυτό είναι το τελευταίο βήμα του Ενορχηστρωτή AutoML και ο σκοπός του είναι να παρακολουθεί το πείραμα Katib που δημιούργησε το προηγούμενο βήμα.

Listing 4.14: The class definition and *do\_run* method of the *MonitorKatibExperiment* step

```

1 class MonitorKatibExperiment(Step):
2     """Wait for a Katib experiment to complete.
3
4     Ins:
5         katib_experiment_name (str): Name of the Katib experiment to monitor.
6     """
7     name = "monitor-katib-experiment"
8
9     def do_run(self, katib_experiment_name: str):
10        """Implementation of MonitorKatibExperiment."""
11        from kale.common import katibutils
12
13        katibutils.wait_for_hptuning_experiment(katib_experiment_name)

```

Όπως είναι φανερό παραπάνω, η μέθοδος *do\_run* του *MonitorKatibExperiment* ουσιαστικά περιμένει το πείραμα Katib να τελειώσει την εκτέλεση του. Το κάνει αυτό καλώντας την βοηθητική συνάρτηση *wait\_for\_hptuning\_experiment* από το δομοστοιχείο *common.katibutils* του Kale. Παρακάτω φαίνεται ο κώδικας της συνάρτησης:

Listing 4.15: Η βοηθητική συνάρτηση *wait\_for\_hptuning\_experiment* του δομοστοιχείου *kale.common.katibutils*

```

1 def wait_for_hptuning_experiment(experiment_name: str):
2     """Wait for an HP Tuning experiment to succeed.
3
4     Args:
5         experiment_name (str): Name of the HP Tuning experiment.
6
7     Returns:
8         tuple(str, str, str): Status, Condition reason, Condition message.
9     """
10    def sleep_with_progress(total, interval, msg):
11        for i in range(0, total // interval):
12            log.info("%s %d...", msg, total - i * interval)
13            time.sleep(interval)
14
15    while True:
16        log.newline(2)
17        log.info("Watching for HP Tuning experiment: '%s'",
18                experiment_name)
19        sleep_with_progress(30, 5, "Checking status in")
20
21        experiment = get_experiment(experiment_name, podutils.get_namespace())
22        status = get_experiment_status(experiment["status"])

```

```

23     log.info("Experiment status: %s", status)
24     if status[0] not in EXPERIMENT_FINAL_STATES:
25         continue
26     return status

```

### 4.3 Οι Ροές Διαμόρφωσης

Σε αυτή την ενότητα, θα περιγράψουμε την αρχιτεκτονική των Ροών Διαμόρφωσης. Πρόκειται για διοχετεύσεις τις οποίες δημιουργεί ο Ενορχηστρωτής AutoML κατά την διάρκεια του βήματος *RunMetaLearningConfigurations*, έτσι ώστε να εκπαιδεύσει μοντέλα παράλληλα. Κάθε Ροή Διαμόρφωσης σε ένα πείραμα AutoML υλοποιεί μία από τις διαμορφώσεις διοχέτευσης που παράγει το *auto-sklearn*.

Αυτές οι διοχετεύσεις ουσιαστικά αποτελούνται από τρία βήματα, και κάθε βήμα αντιστοιχεί σε ένα συγκεκριμένο τμήμα μιας ροής εργασίας μηχανικής μάθησης. Παρακάτω παρατίθεται μια περίληψη της λειτουργίας του κάθε βήματος:

1. **run-sklearn-transformer**: Αυτό το βήμα υλοποιεί τον προ-επεξεργαστή που περιγράφει μια διαμόρφωση διοχέτευσης. Ο σκοπός του είναι να επεξεργαστεί το σύνολο δεδομένων εισόδου (τόσο το σύνολο εκπαίδευσης όσο και το σύνολο εξέτασης) και να το φέρει σε μια κατάσταση που θα επιτρέψει στο μοντέλο να εκπαιδευτεί και να εξετασθεί η απόδοση του πάνω σε αυτό το σύνολο δεδομένων.
2. **train-sklearn-estimator**: Αυτό το βήμα ουσιαστικά υλοποιεί την αρχιτεκτονική του μοντέλου που περιγράφει μια διαμόρφωση διοχέτευσης. Ο σκοπός του είναι να παράξει ένα εκπαιδευμένο μοντέλο χρησιμοποιώντας το επεξεργασμένο σύνολο δεδομένων εκπαίδευσης που παράγει το προηγούμενο βήμα.
3. *infer-sklearn-predictor*: Αυτό το βήμα χρησιμοποιεί το επεξεργασμένο σύνολο δεδομένων εκπαίδευσης του βήματος *run-sklearn-transformer* για να ελέγξει την απόδοση του μοντέλου που παράχθηκε στο βήμα **train-sklearn-estimator** χρησιμοποιώντας την συνάρτηση μετρικής που ο χρήστης παρέιχε. Όταν τα σκορ μετρικής από αυτό το βήμα έχουν τελικά παραχθεί, ολόκληρη η Ροή Διαμόρφωσης ολοκληρώνει την εκτέλεση της.

Παρακάτω παρουσιάζουμε ένα παράδειγμα μιας Ροής Διαμόρφωσης που έχει ολοκληρώσει την λειτουργία της, όπως φαίνεται στην διεπαφή χρήστη του KFP.

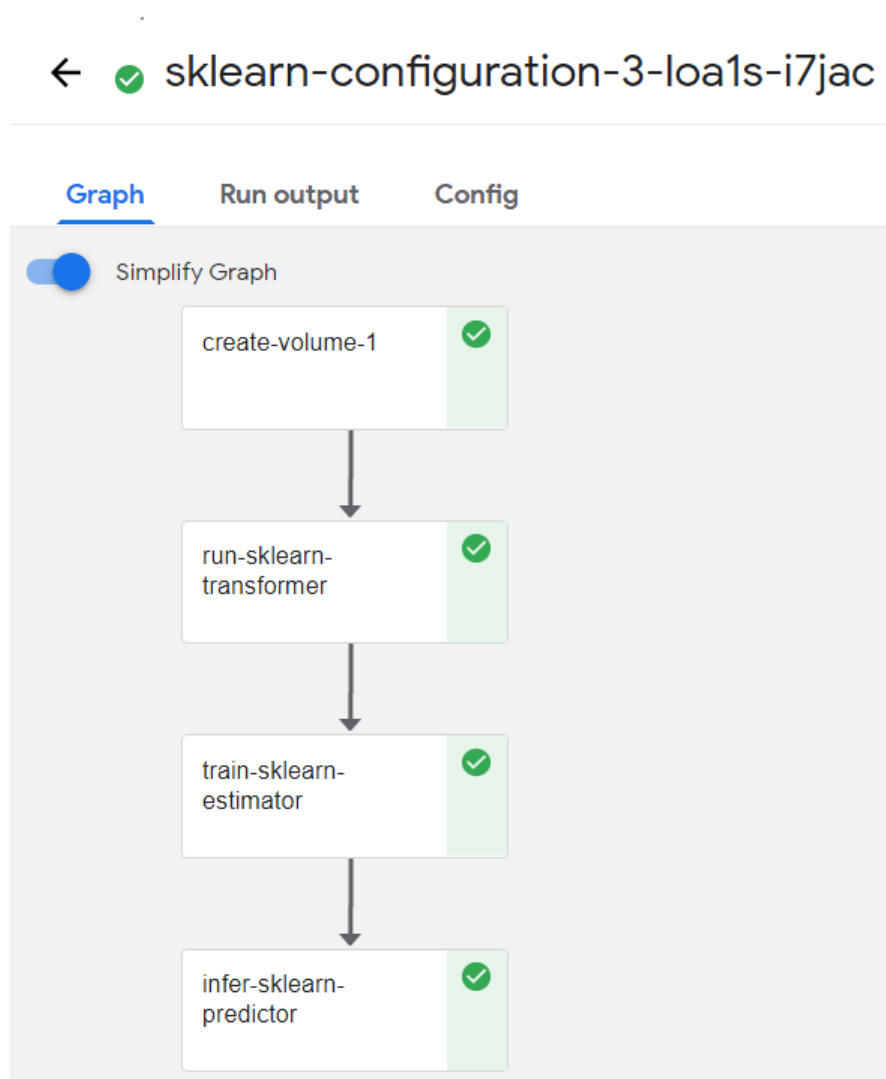


Figure 4.2: Παράδειγμα ολοκληρωμένης Ροής Διαμόρφωσης στην διεπαφή χρήστη του KFP

Όπως αναφέραμε σε στην υποενότητα 4.2.3, η συνάρτηση διοχέτευσης που χρησιμοποιείται σαν μακέτα για τις Ροές Διαμόρφωσης είναι η `sklearn_train_predict()` από το δομοστοιχείο `kale.ml.pipelines`. Στην επόμενη υποενότητα παραθέτουμε και αναλύουμε αυτήν την συνάρτηση διοχέτευσης.

### 4.3.1 Η Συνάρτηση Διοχέτευσης `sklearn_train_predict`

Η συνάρτηση `sklearn_train_predict` είναι μια συνάρτηση διοχέτευσης Python, γραμμένη στην γλώσσα συγκεκριμένου τομέα του Kale (2.5.4), η οποία ουσιαστικά περιγράφει την δομή της Ροής Διαμόρφωσης που το δημιουργεί το βήμα `RunMetaLearningConfigurations` (4.2.3). Εφόσον είναι γραμμένη σε γλώσσα ειδικού σκοπού του Kale, χρησιμοποιεί αντικείμενα `Step` (2.5.2) του Kale για να αναπαραστήσει βήματα στην διοχέτευση. Παρακάτω φαίνεται ο κώδικας της συνάρτησης:

Listing 4.16: Η συνάρτηση διοχέτευσης `sklearn_train_predict`

```
1 def sklearn_train_predict(kale_dataset_id="-1", kale_config_id="-1"):
2     """Train and validate a SKLearn model from an AutoML configuration."""
```



```

3     (x_processed,
4     x_test_processed,
5     kale_dataset_id_local,
6     transformer_pipeline) = RunSKLearnTransformer()(
7         ml_assets_marshal_path("configuration.dillpkl"),
8         ml_assets_marshal_path("dataset.dillpkl"),
9         kale_dataset_id,
10        kale_config_id,
11        "{}"
12    )
13    model, kale_model_id = TrainSKLearnEstimator()(
14        ml_assets_marshal_path("configuration.dillpkl"),
15        x_processed,
16        ml_assets_marshal_path("dataset.dillpkl"),
17        kale_dataset_id_local,
18        "{}"
19    )
20    InferSKLearnPredictor()(model,
21                            x_processed,
22                            x_test_processed,
23                            ml_assets_marshal_path("metric.joblib"),
24                            ml_assets_marshal_path("dataset.dillpkl"),
25                            ml_assets_marshal_path("task.dillpkl"),
26                            kale_model_id,
27                            kale_dataset_id_local)

```

Όπως φαίνεται από τον προηγούμενο κώδικα, η διοχέτευση μιας Ροής Διαμόρφωσης που περιγράφει η συνάρτηση *sklearn\_train\_predict*, αποτελείται από τρία βήματα:

1. **RunSKLearnTransformer**
2. **TrainSKLearnEstimator**
3. **InferSKLearnPredictor**

Κάθε ένα από αυτά είναι ένα βήμα Kale που υλοποιεί ένα συγκεκριμένο τμήμα της διοχέτευσης μιας Ροής Διαμόρφωσης. Στις ακόλουθες υποενότητες, θα εκθέσουμε την λειτουργικότητα καθενός από αυτά τα βήματα.

### 4.3.2 Το Βήμα *RunSKLearnTransformer*

Σε αυτή την υποενότητα, θα εκθέσουμε και θα εξηγήσουμε τον μηχανισμό που υλοποιεί το βήμα *RunSKLearnTransformer*. Θα περιγράψουμε πως αυτό το βήμα διαβάζει ως είσοδο μια διαμόρφωση διοχέτευσης και υλοποιεί το βήμα του προ-επεξεργαστή που περιγράφει η διαμόρφωση αυτή. Παρακάτω, εκθέτουμε τον ορισμό κλάσης και τις κύριες μεθόδους του βήματος:

Listing 4.17: Ο ορισμός κλάσης και οι κύριες μέθοδοι του βήματος *RunSKLearnTransformer*

```

1 class RunSKLearnTransformer(PatchMaskedInputsMixin, Step):
2     """Run a SKLearn transformer over a dataset.

```

```
3
4     Ins:
5         configuration
6         dataset
7         kale_dataset_id
8         kale_config_id
9
10    Outs:
11        x_processed
12        x_test_processed
13        kale_dataset_id_local
14        transformer_pipeline
15
16    MLMD Inputs:
17        kale.Configuration
18        kale.Dataset
19
20    MLMD Outputs:
21        kale.Transformer
22    """
23    name = "run-sklearn-transformer"
24    outs = Param.odict({"x_processed", "x_test_processed",
25                       "kale_dataset_id_local", "transformer_pipeline"},
26                      step_name=name)
27
28    def _submit_dataset_artifact(self, dataset):
29        from kale.settings import settings
30        from kale.marshall.utils import strip_marshall_path
31
32        dataset_artifact = dataset.as_artifact()
33        rok_version = self.vars.get("autosnapshot_start")
34        if not dataset.artifact_uri and rok_version:
35            dataset.artifact_uri = rokutils.get_uri_in_version(
36                rok_version, strip_marshall_path(settings.INS["dataset"]))
37        dataset_artifact = dataset.submit_artifact()
38        return dataset_artifact
39
40    def _link_input_artifacts(self, kale_dataset_id, kale_config_id):
41        mlmd = mlmdutils.get_mlmd_instance()
42        if kale_config_id > 0:
43            mlmd.link_artifact_as_input(kale_config_id)
44        if kale_dataset_id > 0:
45            mlmd.link_artifact_as_input(kale_dataset_id)
46
47    def _link_output_artifacts(self):
48        from kale.common.artifacts import Transformer
49
50        mlmd = mlmdutils.get_mlmd_instance()
```

```

51     self.vars["transformer_artifact"] = Transformer(
52         self.name, self.vars["preprocessor"]).submit_artifact()
53     mlmd.link_artifact_as_output(self.vars["transformer_artifact"].id)
54
55     def do_run(self, configuration, dataset, kale_dataset_id, kale_config_id,
56         masked_inputs):
57         """Implementation of RunSKLearnTransformer."""
58         import json
59         import sklearn
60         from kale.ml import utils
61
62         kale_dataset_id = int(kale_dataset_id)
63         kale_config_id = int(kale_config_id)
64         if kale_dataset_id <= 0:
65             kale_dataset_id = self._submit_dataset_artifact(dataset).id
66         self._link_input_artifacts(kale_dataset_id, kale_config_id)
67
68         configuration = utils.patch_configuration(configuration,
69                                                     json.loads(masked_inputs),
70                                                     coerce=True)
71         steps = utils.get_sklearn_steps_list(configuration)
72         self.vars["preprocessor"] = sklearn.pipeline.Pipeline(steps[:-1])
73         x_processed = self.vars["preprocessor"].fit_transform(dataset.features,
74                                                             dataset.targets)
75         x_test_processed = self.vars["preprocessor"].transform(
76             dataset.features_test)
77
78         self._link_output_artifacts()
79         return (x_processed, x_test_processed, kale_dataset_id,
80             self.vars["preprocessor"])

```

Όπως και στην περίπτωση των βημάτων του Εννορηστροπή AutoML, η μέθοδος *do\_run()* είναι ο κύριος κώδικας που τρέχει στο εσωτερικό του βήματος. Επομένως, ας εκθέσουμε μια περίληψη της λογικής που υλοποιεί αυτή η μέθοδος. Η μέθοδος *do\_run*:

1. **Ορίζει τα Artifacts Dataset και AutoMLConfiguration ως εισόδους αυτού του βήματος.** Ο ορισμός αυτός υλοποιείται με χρήση της μεθόδου *\_link\_input\_artifacts* του βήματος.
2. **Διαβάζει την διαμόρφωση διοχέτευσης που δέχεται ως είσοδο.** Αν η τρέχουσα Ροή Διαμόρφωσης είναι μέρος ενός πειράματος Katib, (δημιουργημένη από το βήμα *RunKatibExperiment* του Εννορηστροπή AutoML) τότε ανανεώνουμε την διαμόρφωση διοχέτευσης, που έχει δημιουργηθεί από το auto-sklearn, με τις τιμές που προτείνει το Katib. Στην περίπτωση που η τρέχουσα Ροή Διαμόρφωσης έχει δημιουργηθεί από το βήμα *RunMetaLearningConfigurations*, κρατάμε την διαμόρφωση διοχέτευσης ως έχει. Υλοποιήσαμε αυτήν την απόφαση στην συνάρτηση *patch\_configuration* του δομοστοιχείου *kale.ml.utils*:

Listing 4.18: Η συνάρτηση *patch\_configuration* του δομοστοιχείου *kale.ml.utils*

```

1     def patch_configuration(configuration: Configuration,
2                             overrides: Dict[str, Any],
3                             coerce=False):
4         """Patch a configuration.
5
6         Args:
7             configuration: A suggested configuration extracted by auto-sklearn.
8             overrides (dict): Override configuration values by providing them with
9                 keys matching the last token in the original Configuration keys.
10                Provide an empty dictionary to make this function a no-op.
11            coerce (bool): Convert the override value to the destination type.
12
13        Returns:
14            Configuration: Patched configuration.
15        """
16        log.info("Patching base configuration...")
17        log.info("Base %s", configuration)
18
19        if not overrides:
20            log.info("No input parameters found to patch the base"
21                    " configuration.")
22            return configuration
23
24        log.info("Using the following configs to patch the base"
25                " configuration: %s", overrides)
26        patched_config = copy.deepcopy(configuration)
27        for name, override_value in overrides.items():
28            for key, conf_value in patched_config.get_dictionary().items():
29                if key.split(":")[-1] == name:
30                    if coerce:
31                        override_value = type(conf_value)(override_value)
32                    patched_config[key] = override_value
33        log.info("Patched %s", patched_config)
34        return patched_config
35

```

3. **Μετατρέπει την διαμόρφωση διοχέτευσης σε μία ταξινομημένη λίστα με βήματα.** Για τον σκοπό αυτό υλοποιήσαμε την συνάρτηση `get_sklearn_steps_list` του δομοστοιχείου `kale.ml.utils`:

Listing 4.19: The `get_sklearn_steps_list` function of the `kale.ml.utils` module

```

1     def get_sklearn_steps_list(config: Configuration):
2         """Return a list of steps for sklearn Pipeline."""
3         log.info("Getting SKLearn steps list...")
4         if config.get("classifier:__choice__"):
5             p = SimpleClassificationPipeline(config)
6         elif config.get("regressor:__choice__"):

```

```

7     p = SimpleRegressionPipeline(config)
8     else:
9         raise ValueError("No model found in input configuration")
10    return p.steps

```

4. Αφαιρεί το τελευταίο βήμα που αντιστοιχεί στο μοντέλο και κρατάει την υπόλοιπη λίστα.
5. Δημιουργεί ένα αντικείμενο *sklearn.pipeline Pipeline object* με την λίστα που τώρα περιέχει μόνο βήματα προ-επεξεργασίας.
6. Επεξεργάζεται το σύνολο δεδομένων κάνοντας χρήση των μεθόδων *transform* και *fit\_transform* (2.1.1) του αντικειμένου Pipeline του sklearn.
7. Δημιουργεί ένα **Transformer Artifact (4.4.1.3)** και το ορίζει ως έξοδο του βήματος χρησιμοποιώντας την συνάρτηση *\_link\_output\_artifacts*.
8. **Επιστρέφει το επεξεργασμένο σύνολο δεδομένων για το επόμενο βήμα της Ροής Διαμόρφωσης.**

### 4.3.3 Το Βήμα *TrainSKLearnEstimator*

Σε αυτήν την υποενότητα, θα εκθέσουμε και θα εξηγήσουμε τον μηχανισμό που υλοποιεί το βήμα *TrainSKLearnEstimator*. Γενικά, η μέθοδος *do\_run* του βήματος ακολουθεί την ίδια λογική με την *do\_run* του προηγούμενου βήματος. Θα περιγράψουμε πώς το βήμα *TrainSKLearnEstimator* διαβάζει την διαμόρφωση διοχέτευσης που δέχεται ως είσοδο, κατασκευάζει το μοντέλο που περιγράφει η διαμόρφωση και ύστερα εκπαιδεύει το μοντέλο πάνω στο επεξεργασμένο σύνολο δεδομένων εκπαίδευσης που δημιούργησε το προηγούμενο βήμα. Ας δούμε τον ορισμό κλάσης και τον κώδικα της μεθόδου *do\_run*:

Listing 4.20: Ο ορισμός κλάσης και οι βασικές μέθοδοι του βήματος *TrainSKLearnEstimator*

```

1  class TrainSKLearnEstimator(PatchMaskedInputsMixin, Step):
2      """Train a SKLearn estimator from an AutoML configuration.
3
4      Ins:
5          configuration:
6          x_processed:
7          dataset:
8          kale_dataset_id_local:
9
10     Outs:
11         model: Trained model
12         kale_model_id: MLMD artifact ID of the trained model
13
14     MLMD Inputs:
15         kale.Configuration
16
17     MLMD Outputs:
18         kale.Model
19     """

```

```

20     name = "train-sklearn-estimator"
21     outs = Param.odict(["model", "kale_model_id"], step_name=name)
22
23     def _link_input_artifact(self, kale_dataset_id):
24         mlmd = mlmdutils.get_mlmd_instance()
25         mlmd.link_artifact_as_input(kale_dataset_id)
26
27     def _link_output_artifacts(self, model):
28         from kale.common.artifacts import Model
29
30         mlmd = mlmdutils.get_mlmd_instance()
31         self.vars["kale_model_artifact"] = Model(
32             model, context_name=self.name).submit_artifact()
33         mlmd.link_artifact_as_output(self.vars["kale_model_artifact"].id)
34
35     def do_run(self, configuration, x_processed, dataset,
36               kale_dataset_id_local, masked_inputs):
37         """Implementation of TrainSKLearnEstimator."""
38         import json
39         from kale.ml import utils
40
41         self._link_input_artifact(kale_dataset_id_local)
42
43         configuration = utils.patch_configuration(configuration,
44                                                  json.loads(masked_inputs),
45                                                  coerce=True)
46         self.vars["model"] = utils.get_sklearn_steps_list(configuration)[-1][1]
47
48         # NOTE: The 'model' variable is important, because we use this very
49         # same marshalled model to also start KFSServing servers, which expects
50         # to find 'model.joblib' files.
51         self.vars["model"].fit(x_processed, dataset.targets)
52
53         self._link_output_artifacts(self.vars["model"])
54         return self.vars["model"], self.vars["kale_model_artifact"].id

```

#### 4.3.4 Το Βήμα *InferSKLearnPredictor*

Σε αυτή την υπο-ενότητα θα εκθέσουμε και θα περιγράψουμε τον μηχανισμό που υλοποιεί το βήμα *InferSKLearnPredictor*. Συνοπτικά, η μέθοδος *do\_run* του βήματος αυτού διαβάζει το εκπαιδευμένο μοντέλο που δέχεται ως είσοδο από το προηγούμενο βήμα, και ύστερα ελέγχει το μοντέλο πάνω στο επεξεργασμένο σύνολο δεδομένων εξέτασης που δημιούργησε το βήμα *RunSKLearnTransformer*. Ας δείξουμε τον ορισμό κλάσης και τις κύριες μεθόδους του βήματος:

Listing 4.21: Ο ορισμός κλάσης και οι κύριες μέθοδοι του βήματος *InferSKLearnPredictor*

```

1 class InferSKLearnPredictor(Step):

```

```

2     """Run predictions on a trained model.
3
4     Ins:
5         model
6         x_test_processed
7         metric
8         dataset
9         task
10        kale_model_id
11        kale_dataset_id_local
12
13    MLMD Inputs:
14        kale.Model
15        kale.Dataset
16
17    MLMD Outputs:
18        kale.TensorboardLogs
19    """
20    name = "infer-sklearn-predictor"
21    has_metrics = True
22
23    def _link_input_artifacts(self, kale_dataset_id, kale_model_id):
24        mlmd = mlmdutils.get_mlmd_instance()
25        mlmd.link_artifact_as_input(kale_dataset_id)
26        mlmd.link_artifact_as_input(kale_model_id)
27
28    def _update_model_artifact(self, kale_model_id, metric_name, metric_value):
29        mlmd = mlmdutils.get_mlmd_instance()
30        model = mlmdutils.get_artifact_by_id(kale_model_id)
31        model_metrics = json.loads(model.properties["metrics"].string_value)
32        model_metrics.update({metric_name: metric_value})
33
34        model_name = model.properties["name"].string_value
35        class_name = model_name.split("/")[-1]
36        new_name = "%s/%s" % (mlmd.name or "", class_name)
37
38        mlmdutils.patch_artifact_properties(kale_model_id,
39                                           {"name": new_name,
40                                            "metrics": model_metrics})
41
42    def _produce_report(self, task, dataset, model):
43        from kale.ml import visualizations
44        from tensorboardX import SummaryWriter
45
46        log.info('Logging report to Tensorboard...')
47
48        viz = visualizations.SklearnVisualizer(task=task, dataset=dataset,
49                                               model=model.choice.estimator,

```

```

50         is_fitted=True)
51     report = viz.plot_report()
52     writer = SummaryWriter(log_dir="logs/experiment")
53     for name, figure in report.items():
54         writer.add_figure(name, figure)
55     log.info("Successfully logged report to Tensorboard")
56     self.vars["tb_logs_dir"] = os.path.realpath("logs")
57
58     def _log_tb_logs_artifact(self):
59         from kale.common.artifacts import TensorboardLogs
60
61         rok_version = self.vars.get("autosnapshot_end")
62         uri = self.vars["tb_logs_dir"]
63         if os.path.exists(uri) and rok_version:
64             uri = rokutils.get_uri_in_version(rok_version,
65                                             self.vars["tb_logs_dir"])
66         tb_logs = TensorboardLogs(name=self.name)
67         tb_logs.artifact_uri = uri
68         tb_logs_artifact = tb_logs.submit_artifact()
69
70         mlmd = mlmdutils.get_mlmd_instance()
71         mlmd.link_artifact_as_output(tb_logs_artifact.id)
72
73     def do_run(self, model, x_processed, x_test_processed, metric, dataset,
74              task, kale_model_id, kale_dataset_id_local):
75         """Implementation of InferSKLearnPredictor."""
76         import autosklearn.metrics as metrics
77         from kale.sdk.logging import log_metric
78         from kale.ml.utils import extract_task_type
79
80         self._link_input_artifacts(kale_dataset_id_local, kale_model_id)
81
82         predictions = model.predict(x_test_processed)
83         task_type = extract_task_type(dataset.targets_test, task)
84
85         # 'calculate_score' returns negative values if "lower is better" for
86         # the given metric. For this reason, we will get the absolute number
87         # of the result.
88         metric_value = abs(metrics.calculate_score(dataset.targets_test,
89                                                  predictions, task_type,
90                                                  metric))
91         log.info("Metric '%s': %s", metric.name, metric_value)
92         log_metric(metric.name, metric_value)
93
94         self._update_model_artifact(kale_model_id, metric.name, metric_value)

```



## 4.4 Η Γενεαλογία MLMD και η Παρακολούθηση Κατάστασης των Πειραμάτων AutoML του Kale

Κατά την διάρκεια της ανάπτυξης της διεργασίας AutoML του Kale, θέλαμε να δημιουργήσουμε μια από-άκρη-σε-άκρη γενεαλογία των πειραμάτων AutoML. Ο τελικός στόχος ήταν να επιτρέψουμε στον χρήστη, που καλεί την συνάρτηση `run_automl`, να βλέπει μια ολόκληρη περίληψη του δημιουργημένου πειράματος AutoML. Η περίληψη αυτή περιλαμβάνει:

- **την κατάσταση του Ενορχηστρωτή AutoML**
- **την κατάσταση των Ροών Διαμόρφωσης**
- **την κατάσταση του πειράματος Katib** στην περίπτωση που παρέχεται ένα αντικείμενο εισόδου `tuner` στην συνάρτηση `run_automl`

Είχαμε ως στόχο να μπορούμε να έχουμε εποπτεία όλων των παραπάνω, από μια κλήση συνάρτησης `run_automl` που της παρέχεται το ID της διοχέτευσης του Ενορχηστρωτή AutoML.

Δημιουργώντας μια γενεαλογία στο MLMD, καταφέραμε να παρακολουθήσουμε την κατάσταση ολόκληρου του πειράματος AutoML, ξεκινώντας από το ID του Ενορχηστρωτή AutoML.

### 4.4.1 Artifacts του Kale

Σε αυτήν την ενότητα θα εξετάσουμε τους διαφορετικούς τύπους από Artifacts που δημιουργεί το Kale ώστε να διατηρήσει μια γενεαλογία MLMD για όλη την διάρκεια ενός πειράματος AutoML.

Για να μπορούμε να υποβάλλουμε και να ανανεώνουμε διαφορετικών ειδών MLMD Artifacts, δημιουργήσαμε το δομοστοιχείο `kale.common.artifacts` το οποίο περιέχει ορισμούς κλάσεων για όλα τα Artifacts που θα χρειαστούμε. Θα εκθέσουμε αυτές τις κλάσεις στις ακόλουθες υποενότητες.

#### 4.4.1.1 Η Βασική Κλάση `MLMDArtifact`

Αυτή είναι η βασική κλάση για αντικείμενα που υποβάλλονται στην βάση του MLMD. Άλλες κλάσεις που αντιστοιχούν σε Artifacts του MLMD και των οποίων τα αντικείμενα πρέπει να υποβληθούν στο MLMD, πρέπει να κληρονομούν από αυτή την κλάση και πρέπει να:

1. **έχουν γνωρίσματα `artifact_type_name` και `artifact_property_types`**
2. **υλοποιούν τις ακόλουθες ιδιότητες: `artifact_properties` και `artifact_custom_properties`**
3. **έχουν ένα γνώρισμα `artifact_uri`, που θα έχει τιμή κατά την διάρκεια του χρόνου εκτέλεσης**
4. **υλοποιούν την μέθοδο `submit_artifact`**

Listing 4.22: Η βασική κλάση `MLMDArtifact` που χρησιμοποιείται για την υποβολή Artifacts στο MLMD

```

1 class MLMDArtifact(abc.ABC):
2     """A base class for objects submittable in MLMD,
3
4     Other classes that correspond to MLMD Artifacts and that expect their instances to be
5     submitted in MLMD should subclass this class.
6
7     Example:
8
9     >>> class MyClass(MLMDArtifact):
10         ...
11         artifact_type_name = "kale.MyClass"
12         artifact_property_types = {"prop1": metadata_store_pb2.STRING,
13                                     "prop2": metadata_store_pb2.INT}
14         ...
15         @property
16         def artifact_properties(self):
17             prop1 = metadata_store_pb2.Value(
18                 string_value=self.get_prop1())
19             prop2 = metadata_store_pb2.Value(
20                 int_value=self.compute_prop2())
21             return {"prop1": prop1, "prop2": prop2}
22         ...
23         @property
24         def artifact_custom_properties(self):
25             custom_prop = metadata_store_pb2.Value(
26                 string_value="custom_value")
27             return {"custom_prop": custom_prop}
28         ...
29     >>> instance = MyClass()
30     >>> instance.artifact_uri = "gs://bucket/path/to/blob"
31     >>> artifact = instance.submit_artifact()
32
33 For more information on available property types, see
34 https://github.com/google/ml-metadata/blob/v0.29.0/ml-metadata/proto/metadata\_store.
35 proto#L74-L81
36
37 Attributes:
38     artifact_type_name (str): The name of the ArtifactType
39     artifact_property_types (dict): The mapping of property names and their
40         MLMD types
41     artifact_uri (str): The URI of the Artifact
42     """
43
44 artifact_type_name: str = "kale.Artifact"
45 artifact_property_types: Dict = None
46 artifact_uri: str = None
47
48 @property

```

```

47     @abc.abstractmethod
48     def artifact_properties(self) -> Dict[str, metadata_store_pb2.Value]:
49         """Get the properties of the Artifact."""
50         pass
51
52     @property
53     def artifact_custom_properties(self) -> Dict[str,
54                                         metadata_store_pb2.Value]:
55         """Get the custom properties of the Artifact."""
56         return {}
57
58     def as_artifact(self) -> metadata_store_pb2.Artifact:
59         """Get the Artifact instance corresponding to this object.
60
61         Compose the Artifact instance based on all the information
62         held/computed by the object itself.
63
64         Returns:
65             metadata_store_pb2.Artifact: The generated Artifact
66         """
67         from ml_metadata.proto.metadata_store_pb2 import Artifact
68
69         log.info("Creating ArtifactType '%s'...", self.artifact_type_name)
70         artifact_type = mlmdutils.get_or_create_artifact_type(
71             type_name=self.artifact_type_name,
72             properties=self.artifact_property_types)
73         log.info("ArtifactType '%s' has ID %d", self.artifact_type_name,
74                 artifact_type.id)
75
76         custom_properties = self.artifact_custom_properties.copy()
77         if hasattr(self, "_mlmd_list_index"):
78             custom_properties["list_index"] = metadata_store_pb2.Value(
79                 int_value=self._mlmd_list_index)
80
81         return Artifact(uri=self.artifact_uri or "",
82                        type_id=artifact_type.id,
83                        properties=self.artifact_properties,
84                        custom_properties=custom_properties)
85
86     def submit_artifact(self) -> metadata_store_pb2.Artifact:
87         """Submit self to ML Metadata.
88
89         Returns:
90             metadata_store_pb2.Artifact: The submitted Artifact
91         """
92         artifact = self.as_artifact()
93
94         log.info("Creating '%s' Artifact...", self.artifact_type_name)

```

```

95     artifact.id = mlmdutils.put_artifact(artifact)
96     log.info("Successfully created '%s' Artifact with ID %d",
97             self.artifact_type_name, artifact.id)
98     return artifact
99
100    def assign_list_index(self, list_index: int):
101        """Assign the artifact to a list of artifacts of the same type.
102
103        In a MLMD context, if some artifacts of type X is part of a list, then
104        all the artifacts of the same type belonging to the same context must
105        be part of the same of list (i.e. must have called
106        'assign_list_index')
107
108        Args:
109            list_index (int): The position of the artifact in the list
110        """
111        self._mlmd_list_index = list_index

```

#### 4.4.1.2 To Artifact *AutoMLConfiguration*

Ένας από τους στόχους μας, σχετικά με την γενεαλογία MLMD του πειράματος AutoML, ήταν να έχουμε την δυνατότητα να ανακτούμε πληροφορίες σχετικές με τις προτεινόμενες διαμορφώσεις διοχέτευσης που παράγει το auto-sklearn. Πιο αναλυτικά, θέλαμε να μπορούμε να λαμβάνουμε αυτήν την πληροφορία χρησιμοποιώντας απλά το ID του Εννορησιρωτή AutoML.

Για να το υλοποιήσουμε αυτό, αποφασίσαμε ότι όταν το βήμα *get-metalearning-configuration* του Εννορησιρωτή εκτελείται, πρέπει να δημιουργεί *AutoMLConfiguration Artifacts* (sub-section 4.2.2). Αυτό το βήμα επίσης ορίζει αυτά τα Artifacts ως εξόδους του. Αυτό συμβαίνει υποβάλλοντας *MLMD Events* (2.6) που ορίζουν ότι τα *AutoMLConfiguration Artifacts* είναι έξοδοι του Execution που αντιστοιχεί στο βήμα *get-metalearning-configuration*.

Αυτά τα *AutoMLConfiguration Artifacts* έχουν μια κύρια ιδιότητα που ονομάζεται *model\_data*. Πριν η μέθοδος *do\_run* του βήματος *GetMetaLearningConfigurations* υποβάλλει ένα *AutoMLConfiguration Artifact* στο MLMD, "γεμίζει" αυτήν την ιδιότητα (4.2.2.4) με ένα λεξικό που περιέχει:

1. **Το όνομα της αρχιτεκτονικής του μοντέλου της αντίστοιχης διαμόρφωσης διοχέτευσης.**
2. **Τα ονόματα και τις τιμές των παραμέτρων του μοντέλου.**

Ο ακόλουθος κώδικας περιέχει τον ορισμό της κλάσης, τα γνωρίσματα και τις μεθόδους του *AutoMLConfiguration*:

Listing 4.23: Η κλάση *AutoMLConfiguration* που χρησιμοποιείται για δημιουργία και υποβολή Artifacts που αντιστοιχούν σε διαμορφώσεις διοχέτευσης

```

1 class AutoMLConfiguration(MLMDArtifact):
2     """An AutoML configuration Artifact."""
3     artifact_type_name = "kale.AutoMLConfiguration"

```

```

4 artifact_property_types = {"model_data": metadata_store_pb2.STRING}
5
6 def __init__(self, config_summary: Dict, run_id: str, estimator_name: str):
7     self.config_summary = config_summary
8     self.run_id = run_id
9     self.estimator_name = estimator_name
10
11 @property
12 def artifact_properties(self) -> Dict[str, metadata_store_pb2.Value]:
13     """Get kale.AutoMLConfiguration Artifact properties."""
14     model_data = metadata_store_pb2.Value(
15         string_value=json.dumps(self.config_summary))
16     return {"model_data": model_data}
17
18 @property
19 def artifact_custom_properties(self):
20     """Get kale.AutoMLConfiguration Artifact custom properties."""
21     run_id_prop = metadata_store_pb2.Value(string_value=self.run_id)
22     name_prop = metadata_store_pb2.Value(string_value=self.estimator_name)
23     return {"run_id": run_id_prop,
24           "name": name_prop}

```

#### 4.4.1.3 Άλλα Artifacts

Το Kale επίσης δημιουργεί και ανανεώνει ένα πλήθος από άλλα MLMD Artifacts, κάθε ένα από τα οποία αντιστοιχεί σε μια συγκεκριμένη "οντότητα" της διεργασίας AutoML του Kale. Εφόσον δεν θα εστιάσουμε κυρίως σε αυτά σε αυτήν την διπλωματική εργασία, θα εκθέσουμε σύντομα αυτά τα Artifacts και τον σκοπό τους, στην ακόλουθη λίστα:

- **Dataset:** Ένα Artifact για παρακολούθηση της κατάστασης του συνόλου δεδομένων μηχανικής μάθησης.
- **Model:** Ένα Artifact για αποθήκευση του εκπαιδευμένου μοντέλου που παράγεται κατά το βήμα *train-sklearn-estimator* (4.3.3) μιας Ροής Διαμόρφωσης.
- **Transformer:** Ένα Artifact για αποθήκευση του pre-processor που παράγεται κατά το βήμα *run-sklearn-transformer* (4.3.2) μιας Ροής Διαμόρφωσης.

#### 4.4.2 Το Αντικείμενο *AutoMLEXperiment*

Αυτό το αντικείμενο είναι το κύριο εργαλείο για την παρακολούθηση της κατάστασης και της εξέλιξης του πειράματος AutoML. Δεδομένου του ID του Ενορχηστρωτή AutoML, ένα αντικείμενο *AutoMLEXperiment* φέρνει και απεικονίζει όλες τις πληροφορίες που εκφράζουν την κατάσταση της διαδικασίας AutoML.

Παρακάτω παραθέτουμε τον κώδικα που περιέχει τον ορισμό κλάσης και την μέθοδο-κατασκευαστή του αντικειμένου *AutoMLEXperiment*:

Listing 4.24: Η κλάση *AutoMLEXperiment*

```

1 class AutoMLEXperiment():

```

```

2     """A status tracker for the AutoML process."""
3     def __init__(self, run_id: str):
4         self.automl_orchestrator_run_id = run_id
5         try:
6             kfputils.get_run(run_id)
7         except ApiException as e:
8             if e.status == 404:
9                 raise ValueError("Invalid run ID: '%s'. The run ID you"
10                                " provided does not correspond to a KFP run."
11                                % run_id)
12                 raise RuntimeError("Failed to retrieve KFP run with ID '%s': %s"
13                                    % (run_id, str(e)))
14
15         self.mlmd_client = mlmdutils.get_client()
16         self.context_type_name = "KfpRun"
17         self.context_name = mlmdutils.get_context_name_from_run_id(
18             self.automl_orchestrator_run_id)
19
20         self.context = None
21         self.automl_configuration_artifacts = []
22         self.automl_configuration_run_ids = []
23
24         self._update_all_info()

```

Η μέθοδος-κατασκευαστής της κλάσης *AutoMLExperiment* αρχικοποιεί ένα αντικείμενο με μόνο το **ID του Ενορχηστρωτή AutoML** ως είσοδο. Παρακάτω φαίνεται η λίστα με τα γνωρίσματα ενός αντικειμένου *AutoMLExperiment*:

- **automl\_orchestrator\_run\_id** (str)
- **context\_type\_name** (str)
- **context\_name** (str)
- **mlmd\_client** (ml\_metadata.MetadataStore)
- **context** (ml\_metadata.proto.metadata\_store\_pb2.Context)
- **automl\_configuration\_artifacts** (list)
- **automl\_configuration\_run\_ids** (list)

Για να πληροφορήσει τον χρήστη σχετικά με την κατάσταση του πειράματος AutoML, ένα αντικείμενο *AutoMLExperiment* παρέχει δύο βασικές μεθόδους διεπαφής:

1. **summary**: Ενημερώνει τον χρήστη σχετικά με την κατάσταση του Ενορχηστρωτή AutoML και των Ροών Διαμόρφωσης, και για τα σκορ μετρικής κάθε μιας από τις Ροές Διαμόρφωσης.
2. **list\_configurations**: Ενημερώνει τον χρήστη σχετικά με την αρχιτεκτονική του μοντέλου και των παραμέτρων της κάθε διαμόρφωσης διοχέτευσης που προτείνει το auto-sklearn.

Παρακάτω, θα δώσουμε μια πιο λεπτομερή εικόνα των δύο αυτών μεθόδων διεπαφής. Αλλά πριν το κάνουμε αυτό, θα εκθέσουμε την λειτουργικότητα μίας πολύ χρήσιμης μεθόδου, που ονομάζεται `_update_all_info`:

Listing 4.25: Η βοηθητική μέθοδος `_update_all_info` της κλάσης `AutoMLExperiment`

```

1
2 def _update_all_info(self):
3     self.context = None
4     self.automl_configuration_artifacts = []
5     self.automl_configuration_run_ids = []
6
7     self.context = self.mlmd_client.get_context_by_type_and_name(
8         type_name=self.context_type_name, context_name=self.context_name)
9
10    if not self.context:
11        return
12
13    if "configuration_runs" in self.context.custom_properties:
14        config_run_ids_str = self.context.custom_properties[
15            "configuration_runs"].string_value
16        self.automl_configuration_run_ids = json.loads(config_run_ids_str)
17
18    self.automl_configuration_artifacts = (
19        mlmdutils.get_artifacts_by_context_and_type(
20            context_id=self.context.id,
21            type_name=AutoMLConfiguration.artifact_type_name,
22            sorted=True))

```

Αυτή η μέθοδος φέρνει όλες τις πληροφορίες που είναι σχετικές με την διεργασία AutoML. Αυτό βοηθά στην συνοχή ανάμεσα σε όλων των ειδών τις πληροφορίες που αφορούν την διεργασία. Πιο αναλυτικά, αυτή η μέθοδος ανακτά:

- **το MLMD Context του Ενορχηστρωτή AutoML**
- **τα AutoMLConfiguration Artifacts**
- **τα IDs των Ροών Διαμόρφωσης**

#### 4.4.2.1 Η Μέθοδος `list_configurations`

Αυτή η μέθοδος διεπαφής εκθέτει πληροφορίες σχετικές με τις αρχιτεκτονικές μοντέλων των προτεινομένων διαμορφώσεων διοχέτευσης. Αυτό το καταφέρνει καλώντας την μέθοδο `_update_all_info` που περιγράψαμε νωρίτερα, έτσι ώστε να συλλέξει όλα τα `AutoMLConfiguration Artifacts` (4.4.1.2) που δημιουργήθηκαν κατά το βήμα `get-metalearning-configurations` (4.2.2).

Ας δούμε τον κώδικα της μεθόδου `list_configurations`:

Listing 4.26: Η μέθοδος `list_configurations`

```

1 def list_configurations(self):
2     """Show information about the suggested model configurations."""

```

```

3     self._update_all_info()
4     if not self.automl_configuration_artifacts:
5         print("There are no configurations yet...\n")
6         return
7
8     ml.utils.print_suggested_models(self.automl_configuration_artifacts)

```

Όπως μπορούμε να δούμε από τον κώδικα που εκθέσαμε παραπάνω, η *list\_configurations* τυπώνει όλες τις πληροφορίες που είναι σχετικές με τα μοντέλα των διαμορφώσεων διοχέτευσης χρησιμοποιώντας την βοηθητική συνάρτηση *print\_suggested\_models* του δομοστοιχείου *ml.utils*:

Listing 4.27: Η βοηθητική συνάρτηση *print\_suggested\_models* από το δομοστοιχείο *kale.ml.utils*

```

1 def print_suggested_models(artifacts: List[Artifact]):
2     """Print suggested models given an MLMD artifacts list."""
3     for idx, artifact in enumerate(artifacts, start=1):
4         print("Configuration", idx)
5         model_dict = json.loads(artifact.properties["model_data"].string_value)
6
7         if "name" not in model_dict:
8             raise RuntimeError("AutoMLConfiguration Artifact for"
9                               " Configuration%d doesn't provide a name for"
10                              " the estimator." % idx)
11         print("Estimator:", model_dict["name"])
12
13         if "parameters" not in model_dict:
14             raise RuntimeError("AutoMLConfiguration Artifact for"
15                               " Configuration%d doesn't provide any"
16                              " parameters." % idx)
17         print("Parameters:")
18
19         for param in model_dict["parameters"]:
20             print("    %s: %s" % (param, model_dict["parameters"][param]))
21         print("\n")

```

Η ακόλουθη εικόνα είναι ένα παράδειγμα του πως η μέθοδος *list\_configurations* τυπώνει όλες τις πληροφορίες που είναι σχετικές με τα μοντέλα των διαμορφώσεων διοχέτευσης όταν την καλούμε στο κελί ενός Jupyter Notebook.

#### 4.4.2.2 Η Μέθοδος *summary*

Η μέθοδος *summary* είναι η κύρια μέθοδος διεπαφής της κλάσης *AutoMLExperiment* που δίνει την δυνατότητα στον χρήστη να παρακολουθήσει την κατάσταση του Ενορχηστρωτή AutoML και των Ροών Διαμόρφωσης, καθώς επίσης και των σκορ μετρικής που κάθε Ροή Διαμόρφωσης παράγει.

Προκειμένου να ανακτήσει αυτές τις πληροφορίες, η μέθοδος *summary* χρειάζεται να ανακτήσει τα ID των διοχετεύσεων που συμμετέχουν στην διεργασία AutoML του Kale. Έχει



```
[10]: automl_experiment.list_configurations()

Configuration 1
Estimator: lda
Parameters:
  shrinkage: manual
  tol: 0.0029112721339713465
  shrinkage_factor: 0.11778631707605758

Configuration 2
Estimator: libsvm_svc
Parameters:
  C: 68.60833575985336
  gamma: 0.03905068757955671
  kernel: rbf
  max_iter: -1
  shrinking: False
  tol: 0.02846559747114773

Configuration 3
Estimator: lda
Parameters:
  shrinkage: None
  tol: 0.09570561577075573

Configuration 4
Estimator: libsvm_svc
Parameters:
  C: 13.284077389890939
  gamma: 0.9060908941898536
  kernel: poly
  max_iter: -1
  shrinking: False
  tol: 0.06323853917206591
```

Figure 4.3: Παράδειγμα της εξόδου της μεθόδου `list_configurations`

ήδη αποκτήσει το ID του Ενορχηστρωτή AutoML (χαρακτηριστικό `automl_orchestrator_run_id`). Επομένως, για να ανακτήσει τα IDs των Ροών Διαμόρφωσης, η μέθοδος `summary` καλεί την βοηθητική μέθοδο `_update_all_info` της κλάσης `AutoMLEXperiment`.

Listing 4.28: Η μέθοδος διεπαφής `summary` της κλάσης `AutoMLEXperiment`

```
1 def summary(self):
2     """Show a summary of the AutoML process."""
3     # Retrieve the current status of the AutoML Orchestrator.
4     try:
5         status = kfputils.get_run(
6             self.automl_orchestrator_run_id).run.status
7     except ApiException as e:
8         if e.status == 404:
9             raise ValueError("Invalid run ID: '%s'. The run ID you"
10                              " provided does not correspond to a KFP run."
11                              % self.automl_orchestrator_run_id)
12     raise RuntimeError("Failed to retrieve KFP run with ID '%s': %s"
```

```

13         %(self.automl_orchestrator_run_id, str(e))
14     if not status:
15         print("The AutoML orchestrator has not started yet...")
16         return
17     print("AutoML orchestrator status: %s\n" % status)
18
19     self._update_all_info()
20     if not self.automl_configuration_artifacts:
21         print("There are no configurations yet...\n")
22         return
23
24     self._print_configuration_run_status_counts()
25     if self.automl_configuration_run_ids:
26         self._print_summary_table()

```

Όπως είναι φανερό από το παραπάνω κομμάτι κώδικα, η μέθοδος *summary* τυπώνει, σειριακά, τέσσερις διαφορετικούς τύπους πληροφορίας

1. **Την κατάσταση του Ενορχηστρωτή AutoML**
2. **Ένα μετρητή των Ροών Διαμόρφωσης που έχουν ξεκινήσει την εκτέλεση τους**
3. **Έναν πίνακα HTML που μετράει τον αριθμό των Ροών Διαμόρφωσης που έχουν κάποια συγκεκριμένη κατάσταση.** Για την υλοποίηση αυτού, η μέθοδος *summary* χρησιμοποιεί μια άλλη μέθοδο της κλάσης *AutoMLExperiment*, η οποία ονομάζεται *called \_print\_configuration\_run\_status\_counts*:

Listing 4.29: Η μέθοδος *\_print\_configuration\_run\_status\_counts* της κλάσης *AutoMLExperiment*

```

1     def _print_configuration_run_status_counts(self):
2         """Print the status of each Configuration Run."""
3         # Initially, print the number of started runs,
4         # out of all the suggested Configurations.
5         num_of_configs = len(self.automl_configuration_artifacts)
6         num_of_config_runs = len(self.automl_configuration_run_ids)
7
8         print("%d/%d Configuration runs have started.\n"
9               % (num_of_config_runs, num_of_configs))
10
11        # Then, print a table that shows how many runs
12        # are currently "Running", "Failed", "Succeeded" etc...
13        status_counts = {"Running": 0}
14        for status in kfputils.KFP_RUN_FINAL_STATES:
15            status_counts[status] = 0
16
17        for run_id in self.automl_configuration_run_ids:
18            try:
19                status = kfputils.get_run(run_id).run.status
20            except ApiException as e:
21                if e.status == 404:

```

```

22         raise ValueError("Invalid run ID: '%s'."
23                            " The run ID you"
24                            " provided does not"
25                            " correspond to a KFP"
26                            " run." % run_id)
27         raise RuntimeError("Failed to retrieve KFP run"
28                            " with ID '%s':"
29                            " %s" % (run_id, str(e)))
30
31     if status not in status_counts:
32         status_counts[status] = 0
33     status_counts[status] += 1
34
35     headers = ["Status", "Count"]
36
37     if utils.is_ipython():
38         # If we are running in a Jupyter Notebook we display
39         # the table in HTML format.
40         from IPython.core.display import display, HTML
41         table = tabulate.tabulate(
42             tabular_data=list(status_counts.items()),
43             headers=headers,
44             tablefmt="html")
45         display(HTML(table))
46     else:
47         table = tabulate.tabulate(
48             tabular_data=list(status_counts.items()),
49             headers=headers)
50         print(table)
51     print("\n")

```

Αυτή η μέθοδος αρχικά τυπώνει τον αριθμό των Ροών Διαμόρφωσης που έχουν ξεκινήσει την εκτέλεση τους, από όλες τις προτεινόμενες διαμορφώσεις διοχέτευσης. Ύστερα, τυπώνει έναν πίνακα που δείχνει πόσες από αυτές τις Ροές Διαμόρφωσης έχουν μια συγκεκριμένη κατάσταση, όπως για παράδειγμα: **Running**, **Failed** ή **Succeeded**. Όπως και η `run_automl()` (4.1), αυτή η μέθοδος είναι φτιαγμένη επίσης για να εκτελείται μέσα σε ένα Jupyter Notebook, σε περιβάλλον ενός Jupyter Server του KubeFlow (2.4.2). Αν αυτό συμβαίνει, τότε ο πίνακας τυπώνεται σε μορφή HTML, χρησιμοποιώντας το δομοστοιχείο `IPython.core.display` [32]. Αλλιώς, τυπώνεται σε μορφή απλού κειμένου, χρησιμοποιώντας το πακέτο `tabulate` [33] της Python.

4. **Έναν πίνακα σύνοψης που περιέχει την κατάσταση και τα σκορ μετρικής κάθε μιας Ροής Διαμόρφωσης.** Για να το υλοποιήσει αυτό, η `summary` χρησιμοποιεί άλλη μια μέθοδο της κλάσης `AutoMLExperiment`, που ονομάζεται `_print_summary_table`. Το ακόλουθο κομμάτι κώδικα δείχνει τον κώδικα της βοηθητικής μεθόδου `_print_summary_table`:

Listing 4.30: Η μέθοδος `_print_summary_table` της κλάσης `AutoMLExperiment`

```
1 def _print_summary_table(self):
2     """Print a summary table for the configuration runs."""
3     tabular_data = []
4
5     # This will be the header of the metrics column. When the first
6     # (metric_name, score) pair is received for a configuration run,
7     # the header will be changed to "Metric (<metric_name>)".
8     metric_header = "Metric"
9
10    for i, run_id in enumerate(self.automl_configuration_run_ids):
11        # Get the status of the configuration run that has started.
12        try:
13            status = kfputils.get_run(run_id).run.status
14        except ApiException as e:
15            if e.status == 404:
16                raise ValueError("Invalid run ID: '%s'."
17                                   " The run ID you"
18                                   " provided does not"
19                                   " correspond to a KFP"
20                                   " run." % run_id)
21            raise RuntimeError("Failed to retrieve KFP run"
22                               " with ID '%s':"
23                               " %s" % (run_id, str(e)))
24        metric = "-"
25
26        # Get the MLMD Context of the configuration run, if exists.
27        context_name = mlmdutils.get_context_name_from_run_id(run_id)
28        context = self.mlmd_client.get_context_by_type_and_name(
29            type_name=mlmdutils.RUN_CONTEXT_TYPE_NAME,
30            context_name=context_name)
31
32        if not context:
33            tabular_data.append([i + 1, run_id, status, metric])
34            continue
35
36        # The Model's ArtifactType may not exist yet and be created
37        # during this AutoML experiment.
38        try:
39            artifacts = \
40                mlmdutils.get_artifacts_by_context_and_type(
41                    context_id=context.id,
42                    type_name=Model.artifact_type_name)
43        except NotFoundError:
44            artifacts = []
45
46        if not artifacts:
47            tabular_data.append([i + 1, run_id, status, metric])
```

```

48         continue
49
50     metric_dict = json.loads(
51         artifacts[0].properties["metrics"].string_value)
52     if not metric_dict:
53         tabular_data.append([i + 1, run_id, status, metric])
54         continue
55
56     metric_name = list(metric_dict.keys())[0]
57     metric = metric_dict[metric_name]
58     tabular_data.append([i + 1, run_id, status, metric])
59
60     # Change the header of the Metric column to also include the
61     # name of the metric.
62     metric_header = "Metric (%s)" % metric_name
63
64     headers = ["#", "KFP Run", "Status", metric_header]
65     client = kfputils._get_kfp_client()
66
67     # If we are running in a Jupyter Notebook we can turn
68     # the displayed Run IDs into clickable links.
69     if utils.is_ipython():
70         from IPython.core.display import display, HTML
71
72         # Get the table in HTML format.
73         table = tabulate.tabulate(tabular_data=tabular_data,
74                                 headers=headers,
75                                 tablefmt="html")
76
77         # Replace each configuration run ID string with HTML code that
78         # links to the KFP UI.
79         for run_id in self.automl_configuration_run_ids:
80             link = ("%s/#/runs/details/%s" %
81                   (client._get_url_prefix(), run_id))
82
83             html = ('<a href="%s" target="_blank">%s</a>'
84                   % (link, run_id))
85
86             table = table.replace(run_id, html)
87
88         display(HTML(table))
89     else:
90         # Get the table in regular format.
91         table = tabulate.tabulate(tabular_data=tabular_data,
92                                 headers=headers)
93
94         print(table)
95     print("\n")

```

Αυτή η μέθοδος τυπώνει έναν πίνακα που δείχνει τον αριθμοδείκτη, το ID, την κατάσταση και το σκορ μετρικής για κάθε Ροή Διαμόρφωσης. Ομοίως με την περίπτωση της `_print_configuration_run_status_counts()`, αν η `_print_summary_table()` εκτελείται σε Jupyter Notebook, μέσα σε περιβάλλον Jupyter Server του Kubeflow, τότε ο πίνακας τυπώνεται σε μορφή HTML, χρησιμοποιώντας το δομοστοιχείο `IPython.core.display` [32].

Η εικόνα 4.4 είναι ένα παράδειγμα του πως η μέθοδος `summary` τυπώνει τις καταστάσεις του Ενορχηστρωτή AutoML και των Ροών Διαμόρφωσης, όπως επίσης τα σκορ μετρικής που κάθε Ροή Διαμόρφωσης παράγει.

```
[15]: automl_experiment.summary()
AutoML orchestrator status: Running

4/4 Configuration runs have started.
```

Status	Count
Running	2
Succeeded	2
Skipped	0
Failed	0
Error	0

#	KFP Run	Status	Metric (accuracy)
1	9232db8b-a9d9-49d7-858a-efae9f05ccf4	Succeeded	0.9955555555555555
2	0119dd00-68c1-424d-ab1e-7be6df7bbc19	Succeeded	0.9911111111111112
3	4cf9f27f-d8b9-4541-999f-183a82caf927	Running	-
4	0b6c5de4-772c-45c2-9d30-8e6e607c7ecb	Running	-

Figure 4.4: Παράδειγμα κλήσης της μεθόδου `summary` μέσα σε ένα κελί Jupyter Notebook

Αυτό που είναι επίσης σημαντικό σχετικά με την εκτέλεση της μεθόδου `summary` μέσα σε ένα Jupyter Notebook, είναι το γεγονός ότι κάθε ID Ροής Διαμόρφωσης είναι σύνδεσμος στην σελίδα της διεπαφής χρήση του KFP που αντιστοιχεί στην Ροή Διαμόρφωσης αυτή. Η εικόνα 4.5 δείχνει την σελίδα της διεπαφής χρήση του KFP που εμφανίζεται όταν κάνουμε κλικ πάνω στο πρώτο ID του παραδείγματος της εικόνας 4.4.

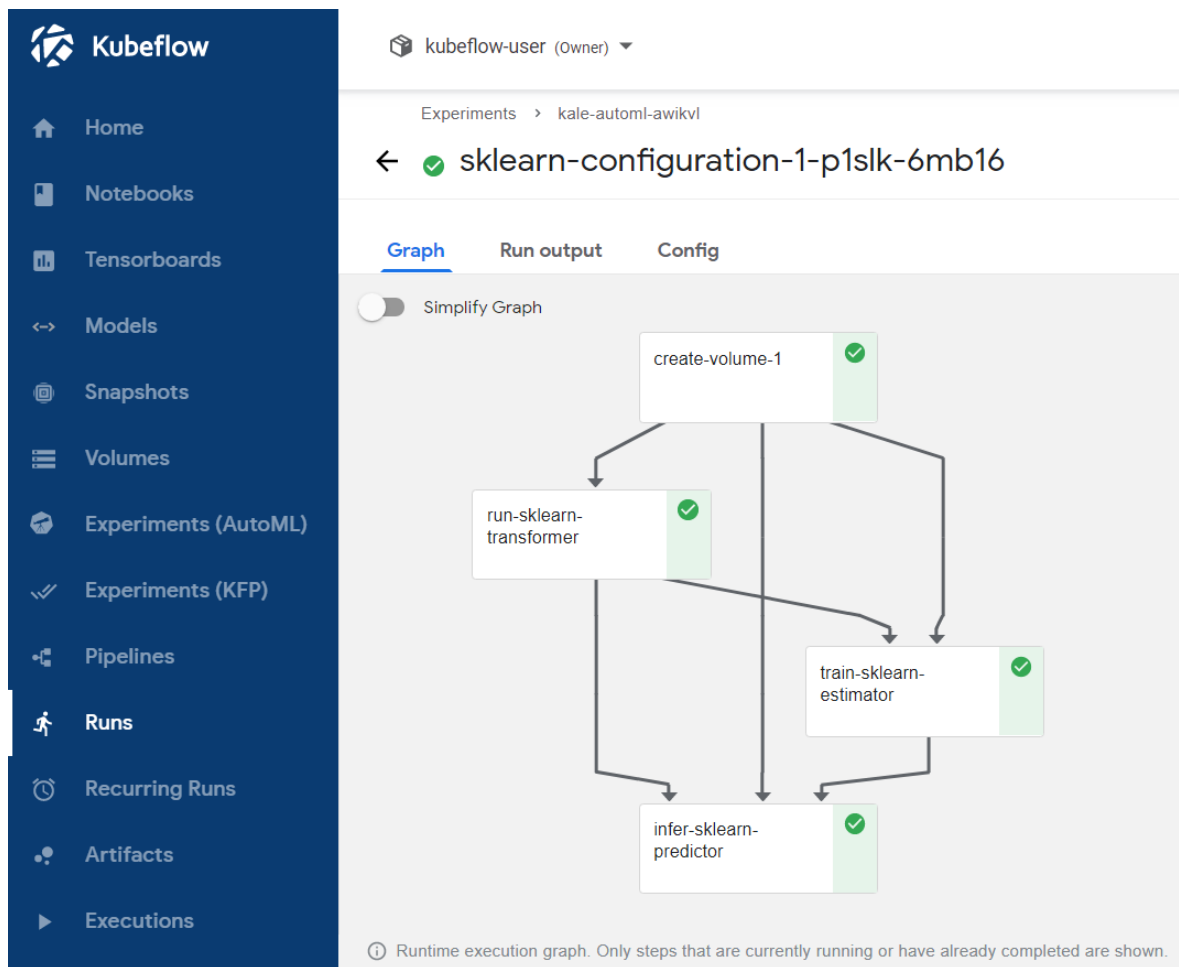


Figure 4.5: Η σελίδα της διεπαφής χρήστη του KFP που αντιστοιχεί σε μια Ροή Διαμόρφωσης





## Chapter 5

# Αξιολόγηση

---

Σε αυτό το κεφάλαιο, θα αξιολογήσουμε την απόδοση του μηχανισμού μας και θα τον συγκρίνουμε με την πιο πρόσφατη έκδοση του *auto-sklearn* (αυτή τη στιγμή η 0.12.7), που είναι η τελευταία λέξη της τεχνολογίας στο πεδίο των βιβλιοθηκών πειραμάτων AutoML. Πιο συγκεκριμένα, μετρήσαμε την επίδοση του μηχανισμού μας και του *auto-sklearn* σε έναν αριθμό πολύ γνωστών σετ δεδομένων μηχανικής μάθησης. Δοκιμάσαμε το μηχανισμό μας σε σετ δεδομένων που χρησιμοποιούνται για εργασίες τόσο παλινδρόμησης, όσο και ταξινόμησης, και τα οποία ποικίλλουν σε μέγεθος.

Για την αξιολόγηση του μηχανισμού μας, συγκριθήκαμε με το μηχανισμό του *auto-sklearn* **τόσο με όσο και χωρίς το μηχανισμό κατασκευής ενωμένων μοντέλων** που περιλαμβάνεται στο *auto-sklearn*. Τα πειράματα με κατασκευή ενωμένων μοντέλων, ανάλογα με το σετ δεδομένων, τείνουν να χρειάζονται περισσότερο χρόνο για να παράξουν ένα αποδοτικό τελικό μοντέλο από το μέσο πείραμα Kale-AutoML, το οποίο κατά την αξιολόγησή μας διήρκεσε περίπου τριάντα λεπτά κατά μέσο όρο.

Μείναμε ευχαριστημένοι διαπιστώνοντας ότι ο μηχανισμός μας αποδίδει εξαιρετικά καλά για όλα τα σετ δεδομένων που χρησιμοποιήσαμε, και **σε κάποιες περιπτώσεις ο μηχανισμός μας είναι καλύτερος από το *auto-sklearn* και με και χωρίς την κατασκευή ενωμένων μοντέλων** ενεργοποιημένη.

## 5.1 Εργαλεία, Μεθοδολογία και Περιβάλλον

Εκτελέσαμε τα πειράματά μας σε ένα περιβάλλον Jupyter Lab μιας εγκατάστασης MiniKF (2.4.5), σε ένα μονό κόμβο στο Google Cloud Platform [34]. Πιο συγκεκριμένα, ο κόμβος ήταν εξοπλισμένος με μία CPU 16 πυρήνων (Intel Xeon 2.3 GHz - Broadwell) μαζί με 60 GB RAM. Ο Jupyter Server (2.4.2) που χρησιμοποιήσαμε για την εκτέλεση των πειραμάτων μας προσάρτησε 4 πυρήνες της CPU μαζί με GB RAM και 4GB αποθηκευτικού χώρου για το χώρο εργασίας, τα οποία συμπεράναμε ότι ήταν επαρκή.

Η σουίτα αξιολόγησής μας αποτελείται από 5 επιβλεπόμενα σετ δεδομένων μάθησης τα οποία εκθέτουμε στον πίνακα 5.1.

Κάθε ένα από τα Kale-AutoML πειράματά μας εκτελέστηκε χρησιμοποιώντας τη συνάρτησή μας `run_automl` (4.1) του API, και χρησιμοποίησε **πέντε διαμορφώσεις διοχέτευσης** (`number_of_configurations=5`) ενώ **το πολύ δύο διαμορφώσεις διοχέτευσης** μπορούσαν να εκτελεστούν παράλληλα (`max_parallel_configurations=2`). Παρόμοια, το πείραμα Katib

όνομα	Εργασία Μηχανικής Μάθησης	Αναλογία Εκπαίδευσης/Δοκιμής
MNIST (35)	ταξινόμηση πολλαπλών κλάσεων	80/20
CIFAR-10 (36)	ταξινόμηση πολλαπλών κλάσεων	80/20
GERMAN CREDIT DATA (37)	δυσιαδική ταξινόμηση	75/25
WINE QUALITY (38)	παλινδρόμηση	75/25
DIABETES (39)	παλινδρόμηση	75/25

Table 5.1: Τα σετ δεδομένων που χρησιμοποιήθηκαν για την αξιολόγηση του AutoML μηχανισμού μας

που υλοποιεί το κομμάτι βελτιστοποίησης υπερπαραμέτρων της διεργασίας μας Kale-AutoML εκτέλεσε **πέντε δοκιμές** ενώ **το πολύ δύο δοκιμές Katib** μπορούσαν να εκτελεστούν παράλληλα.

Για τα αντίστοιχα πειράματα auto-sklearn, χρησιμοποιήσαμε τα αντικείμενα API *AutoSklearnClassifier* και *AutoSklearnRegressor* του auto-sklearn. Οι διαμορφώσεις παραμέτρων εισόδου που παρείχαμε σε αυτά τα αντικείμενα παραθέτονται στον πίνακα 5.2.

παράμετρος	auto-sklearn	auto-sklearn + ενωμένο μοντέλο
time_left_for_this_task (sec)	2000	3000
per_run_time_limit (sec)	360	360
initial_configurations_via_metalearning	5	5
ensemble_size	0	5
ensemble_nbest	0	5
max_models_on_disc	50	50
seed	1	1
memory_limit (MB)	3072	3072

Table 5.2: Οι παράμετροι εισόδου για τα αντικείμενα *AutoSklearnClassifier* και *AutoSklearnRegressor* που χρησιμοποιήσαμε στα πειράματά μας με το auto-sklearn

## 5.2 Αποτελέσματα

Παρουσιάζουμε τις μετρήσεις μας στον πίνακα 5.3. Εκτελέσαμε όλα τα πειράματα σειρικά και όχι παράλληλα ώστε να αποφύγουμε τυχόν υπερβολική χρήση της CPU ή της RAM.

Σετ Δεδομένων	Μετρητικό	auto-sklearn	auto-sklearn + ενωμένο μοντέλο	Kale-AutoML
MNIST	ακρίβεια	0.885	0.942	<b>0.95</b>
CIFAR-10	ακρίβεια	0.25	<u>0.31</u>	0.25
GERMAN CREDIT DATA	ακρίβεια	0.776	<u>0.78</u>	<b>0.78</b>
WINE QUALITY	μέσο τετραγωνισμένο σφάλμα	0.506	0.478	<b>0.457</b>
DIABETES	μέσο τετραγωνισμένο σφάλμα	42.962	<u>42.539</u>	42.902

Table 5.3: Τα αποτελέσματα μετρικής για όλα τα πειράματα.

Μείναμε ικανοποιημένοι βλέποντας ότι στις περιπτώσεις των σετ δεδομένων *MNIST*, *GERMAN CREDIT DATA* και *WINE QUALITY* η διεργασία μας Kale-AutoML **βαθμολογείται εξίσου ή και υψηλότερα από το μηχανισμό του auto-sklearn**. Αποδίδουμε αυτή τη βελτίωση στην ενσωμάτωση του μηχανισμού μας με το Katib, το οποίο εκτελεί βελτιστοποίηση υπερπαραμέτρων στο ήδη εκπαιδευμένο μοντέλο μας με την υψηλότερη βαθμολόγηση.

Χωρίς την ενσωμάτωση του Katib, ο μηχανισμός μας έχει τα ίδια αποτελέσματα με το `auto-sklearn` χωρίς την τεχνική κατασκευής ενωμένων μοντέλων.

Παρατηρούμε ότι, για την περίπτωση του σετ δεδομένων *CIFAR-10*, ο μηχανισμός κατασκευής ενωμένων μοντέλων του `auto-sklearn` επιδεικνύει σημαντική βελτίωση στις βαθμολογίες δοκιμών σε σύγκριση τόσο με τον κανονικό μηχανισμό μεταμάθησης του `auto-sklearn` όσο και με τον μηχανισμό μας. Το *CIFAR-10* είναι το μεγαλύτερο σετ δεδομένων στη σουίτα μας, και αυτό οδήγησε στην αποτυχία τριών από τις πέντε διαμορφώσεις εκτέλεσης στο AutoML πείραμά μας. Ο λόγος για την αποτυχία ήταν ότι ο κλωνοποιημένος λογικός δίσκος χώρου εργασίας που προσάρτησε το Kale σε κάθε βήμα αυτών των Ροών Διαμόρφωσης δεν ήταν επαρκής για να αποθηκεύσει το προεπεξεργασμένο σετ δεδομένων που παράχθηκε από κάποια από τα βήματα `run-sklearn-transformer` (4.3.2).



# Συμπερασματικά Σχόλια

---

Το ταξίδι μας έφτασε στο τέλος του. Σε αυτό το κεφάλαιο, θα επαναδιατυπώσουμε τις συνεισφορές μας και θα συνοψίσουμε τι προσφέρει ο μηχανισμός μας. Τέλος, θα κλείσουμε αυτή τη διπλωματική εργασία αναφέροντας τι μελλοντικό έργο μπορεί να γίνει ώστε να εμπλουτιστεί ο μηχανισμός μας και να φτάσει στο σύνολο των δυνατοτήτων του.

## 6.1 Μια Ανακεφαλαίωση του Μηχανισμού μας

Σχεδιάσαμε, υλοποιήσαμε και αξιολογήσαμε τον μηχανισμό AutoML του Kale. Ας παρουσιάσουμε μια ανακεφαλαίωση του τι προσφέρει για ακόμα μια φορά:

- Εκκινεί πειράματα AutoML στο Kubeflow με μία μόνο κλήση συνάρτησης του API.
- Επιτρέπει στους χρήστες να παρακολουθούν την κατάσταση του πειράματος και των μετρικών αποτελεσμάτων που παράγει μέσω ενός απλού αντικειμένου του API.
- Αξιοποιεί την κατανεμημένη φύση του Κυβερνήτη και το μηχανισμό ενορχήστρωσης διοχετεύσεων του Kale για να καταναίμει τα τμήματα του πειράματος που μπορούν να παραλληλοποιηθούν.
- Χρησιμοποιεί το μηχανισμό μεταμάθησης του *auto-sklearn* ώστε να "warmstart" τη διεργασία και το μηχανισμό βελτιστοποίησης υπερπαραμέτρων του Katib για να προσαρμόσει με ακρίβεια το πιο αποδοτικό μοντέλο.
- Αξιοποιεί το μηχανισμό απόδοσης εκδόσεων σε δεδομένα του Kale για να κάνει κάθε ένα εκπαιδευμένο μοντέλο εύκολα αναπαραζήσιμο και προσβάσιμο ακόμα και μετά το πέρας όλου του πειράματος.

## 6.2 Μελλοντικό Έργο

Μπορούμε τελικά να κλείσουμε αυτή την εργασία με τις μελλοντικές κατευθύνσεις έρευνας του μηχανισμού μας. Σχεδιάζουμε να τις ακολουθήσουμε ενεργά κατά τους επόμενους μήνες ή χρόνια.

- Επέκταση του αντικειμένου μας API *AutoMLExperiment* (4.4.2) για να παρακολουθεί επίσης την κατάσταση του *Katib Experiment*. Προς το παρόν παρακολουθεί μόνο την κατάσταση των Ενορχηστρωτή *AutoML* και των Ροών Διαμόρφωσης.
- Ενσωμάτωση του μηχανισμού κατασκευής ενωμένων μοντέλων του *auto-sklearn* στον *Kale-AutoML* μηχανισμό μας για την παραγωγή ακόμα ισχυρότερων τελικών μοντέλων.
- Επέκταση της κλάσης *Dataset* (4.1.1) ώστε να περιλαμβάνει ένα σετ αξιολόγησης μαζί με τα σετ εκπαίδευσης και δοκιμής που ήδη περιέχει.
- Παροχή στους χρήστες της δυνατότητας να περιορίζουν το χρόνο εκτέλεσης κάθε Ροής Διαμόρφωσης μέσω του *run\_automl* API του *Kale*, όπως ακριβώς στην περίπτωση του *auto-sklearn* API.

## Chapter 7

### Introduction

---

In this first chapter, we outline the scope of our work. We provide a brief overview of the task at hand and we illustrate the gap that there is to fill. Then, we go over the existing approaches, highlighting their offerings and their drawbacks. Moving on, we give a high-level overview of the mechanism we built. Finally, we present the structure of this thesis.

#### 7.1 Problem Statement

Launching a machine learning product generally requires large amounts of data and a sufficient theoretical and practical knowledge of machine-learning techniques and model architectures that can be applied for different tasks. For this reason, creating and deploying machine learning models that work well is a hard task.

At its core, every data scientist needs to solve these fundamental problems of deciding which machine learning algorithm to use on a given dataset, whether and how to pre-process its features, and what values to chose for its hyper-parameters. Finding a good solution for these problems requires both time and computational resources, since one has to experiment with a number of feature-preprocessing techniques, model architectures and hyper-parameter values in order to find a combination that works sufficiently well. This is the problem of *AutoML*, that is: *finding a model that produces accurate test set predictions for a new dataset within a fixed timeline and computational budget*.

A number of AutoML frameworks, that automate the process of *Combined Algorithm Selection and Hyper-parameter tuning (CASH)*, provide satisfactory solutions to this problem. Among them, the *auto-sklearn* [1] Python library which we used for the implementation of our mechanism and which we will expose analytically in chapter 9. However, **there is a main limitation in these frameworks** and that is the fact that they have been built on top of centralized machine learning libraries (e.g. *auto-sklearn* has been built on top of *scikit-learn* [2]) that are designed to work on a single machine and thus the steps of an AutoML process that can be parallelized end up running sequentially in the host machine. Therefore, such frameworks **are not easily scalable**.

On the other hand, cloud-native machine learning platforms, such as *Kubeflow* [3] which will be the main focus of thesis thesis, **offer orchestration and scalability for machine learning workloads but lack in AutoML tools that leverage meta-learning**

**techniques** to produce satisfactory models.

As it can be deduced from above, **there is a gap between AutoML frameworks and cloud native platforms** that prevents data-scientists from leveraging the advantages of both worlds.

## 7.2 Motivation

In the case of some problems, writing a program that tackles the problem in a satisfactory level can prove to be a difficult task for humans. The problem analysis and solution may render impractical or even impossible. In such cases, machine learning is probably the way to go. Machine learning models can be "fed" with large amounts of data, recognize patterns in them and efficiently solve the task at hand.

Data scientists face the difficult task of bringing data in a good state and then **finding a machine learning model configuration that is suitable for solving the given problem**. On top of that, they have to perform all of the above in a timely and cost-effective manner and be able to produce models that are accurate, easily scalable and deployable.

Data scientists are our target audience in this work. Our main goal is to make the lives a lot easier by providing a solution to the aforementioned problems.

## 7.3 Summary of Existing Solutions

In this section, we briefly present the most notable existing solutions in the field of AutoML.

### 7.3.1 Katib

Katib [4] is Kubeflow's tool for hyper-parameter optimization. The main idea behind it is to define the training procedure inside a container and run this logic multiple times, changing the hyper-parameters of the containerized model each time, via the input arguments of the container entry-point command, until reaching a satisfying set of hyper-parameter values.

Katib supports a number of search algorithms which it uses to find sets of hyper-parameters that satisfy the requirements that the user sets regarding the model's final performance. Some of these algorithms are: *Grid Search* and *Bayesian Optimization*. In addition, Katib supports *ENAS* and *DARTS* neural architecture search algorithms for finding neural network architectures that are bound to work well for a given task. For a more detailed overview of all the different search algorithms that Katib offers, refer to the corresponding section in Katib's official documentation [5].

**One basic shortcoming** of Katib is the fact that the user must choose the model architecture/type and provide the implementation of the model for which Katib will search the hyper-parameter space to find a high-scoring hyper-parameter set. **Choosing the correct model for a specific dataset is already a hard task on its own**. In addition,



hyper-parameter optimization is bound to be ineffective, if the model architecture is not suitable for the given dataset and task.

### 7.3.2 Auto-sklearn

**Auto-sklearn** [1] is a Python library for automated machine learning (AutoML) that **frees a machine learning user from model architecture selection** and hyper-parameter tuning. It leverages recent advantages in Bayesian optimization, meta-learning [6] and ensemble construction and it manages to completely replace any *scikit-learn* ([2]) estimator, for supervised machine learning tasks. For the purpose of this thesis, we leveraged *auto-sklearn*'s meta-learning database and mechanism to create a distributed AutoML process on Kubeflow.

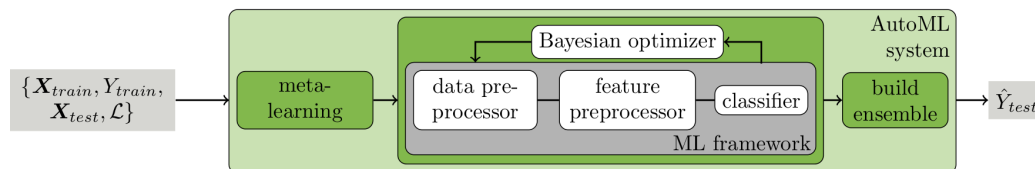


Figure 7.1: *Auto-sklearn in one image*

**The main shortcoming** of auto-sklearn is the fact that, although it starts a number of independent training jobs for different model configurations to find the best-scoring one for a given dataset, it has been built on top of *scikit-learn* which is a centralized machine learning library and is designed to run only on a single machine.

In this thesis, we transfer auto-sklearn's process to *Kubernetes* [7], leveraging its distributed nature, so that we train ML models as parallel pipeline jobs. For a more thorough understanding of *auto-sklearn*'s meta-learning mechanism, refer to chapter 9.

### 7.3.3 AutoGluon

AutoGluon [8] is another Python library that is famous for AutoML tasks. It is essentially an open-source AutoML framework that (similarly to *auto-sklearn*) requires only a few lines of Python code to train highly accurate machine learning models on an unprocessed dataset. It mainly focuses on structured data, such as text, image, and tabular data and it performs advanced data processing, deep learning, and multi-layer model ensembling to maximize its results.

## 7.4 Overview of Our Approach

As we explained above, there is a gap between AutoML toolkits, such as *auto-sklearn*, and cloud native platforms, such as *Kubeflow*, that prevents combining the advantages of both worlds.

Kubeflow [3] is a great platform for orchestrating complex workflows on top of *Kubernetes*. Its self-service nature makes it extremely appealing for data scientists, at it

provides an easy access to advanced distributed jobs orchestration, re-usability of components, Jupyter Notebooks [9], Pipelines [10], rich UIs and more.

In our work, we extended Kale [11], a pipeline orchestration toolkit for Kubeflow, to use auto-sklearn's meta-learning mechanism in order to produce fully-trained, supervised-learning models that are bound to work well for a given dataset, all this while leveraging Kubeflow's components and Kubernetes' distributed nature.

The following **numbered list of steps** describes the mechanism behind the AutoML process that we built for Kale, from the user's perspective:

1. The user provides a **dataset** and the **type of the machine learning task** (classification or regression) as input to Kale's `run_automl()` API function.
2. **With a single function call, the whole process starts** and Kale creates a KubeFlow pipeline to orchestrate the whole process.
3. **The `run_automl` API function returns a Python object to the user** to track the status of the entire AutoML process.
4. Using auto-sklearn's meta-learning kernel, **the orchestrator pipeline computes a list of meta-learning model configurations** that are bound to perform well for the input dataset. These configurations **fully describe entire machine learning pipelines** (e.g. data preprocessors and model architecture).
5. **For each machine learning configuration, the orchestrator pipeline spins up a new pipeline.**
6. **These new pipelines are executed in parallel**, preprocessing the dataset, training the model that the corresponding configuration suggests, and producing test scores **while the orchestrator monitors them.**
7. Once they are all finished, **the orchestrator gathers their scores and selects the model from the pipeline with the best score.**
8. **The orchestrator creates a Katib experiment to further optimize the best-scoring model.**
9. **Kale saves the trained and optimized model and takes a fully reproducible snapshot (8.5.8) of the volume that contains it** so that the user can access it later on and easily reproduce it.

**This thesis mainly focuses on steps 3 to 7 of the Kale-AutoML process.** Nevertheless, we will provide a sufficient analysis of the functionality of the rest of the mechanism as well.

## 7.5 Thesis Structure

The rest of the document is organized as follows:

- **In chapter 8** we provide the theoretical background that is necessary for the reader to understand our work.
- **In chapter 9** we expose our understanding and knowledge of *auto-sklearn*'s meta-learning mechanism.

- **In chapter 10** we analyze the design and the implementation of our mechanism.
- **In chapter 11** we evaluate our work.
- **In chapter 12** we provide a summary of our contributions as well as possible future work directions.



# Background

---

In this chapter we provide the theoretical background necessary for understanding the core practical ideas of the rest of the thesis.

## 8.1 Scikit-learn

*Scikit-learn* (also known as *sklearn*) [2] is an open-source machine learning library for Python [12]. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DB-SCAN, and is designed to interoperate with the Python numerical and scientific libraries *NumPy* [13] and *SciPy* [14].

### 8.1.1 Pipelines

A machine learning pipeline is essentially the product of chaining together a sequence of steps that are involved in the training of a machine learning model. It can be used to automate a machine learning workflow. The pipeline can involve tasks such as:

1. **pre-processing**
2. **feature selection**
3. **classification/regression**

Additionally, more complex applications may need to fit in other necessary steps within this pipeline.

Scikit-learn's *pipeline* module offers a *Pipeline* [15] class object that chains together scikit-learn **transformer** and **estimator** steps into a single ML pipeline. Its main API method, *fit\_transform*, sequentially calls the *fit\_transform* methods of all the steps that exist in the pipeline.

Scikit-learn uses the term "transformers" to refer to objects that implement data/feature preprocessing in scikit-learn, and offer a *fit\_transform* method that probably cleans, reduces, expands or, in general, processes an input dataset. On the other hand, scikit-learn uses the term "estimators" to refer to ML models that can be trained on an input dataset, via a *fit* method, and are able to produce predictions on new unseen data, via a *predict* method.

In the case of a *Pipeline* that only consists of transformers, the *fit\_transform* method of the *Pipeline* actually calls the *fit\_transform* of the first transformer, then feeds its output to the next transformer and so on.

In the general case though, a *Pipeline* consists of transformers and an estimator as a last step. In that case, the *fit\_transform* method of the *Pipeline* object calls the *fit\_transform* methods of the transformer steps sequentially, and then trains the estimator on the transformed data by calling the *fit* method of the estimator step.

## 8.2 OS-Level Virtualization and Containers

### 8.2.1 Overview

Operating-system-level virtualization is an operating system feature in which kernel services allow multiple user-space instances to co-exist as if one is isolated from the others. There is an interesting type of OS-level virtualization instances that are fast, lightweight and easy to use. These instances are called: containers.

Containers look like real, self-standing machines from the point of view of processes that run inside them. They can essentially run in parallel, while sharing the host's OS. This means that each container uses the OS's system call interface and does not need to be subjected to emulation or to run in a virtual machine. This makes containers very lightweight, since they require less overhead in order to be launched, as opposed to full virtualization technologies. From a high-level point of view, this is what differentiates them from virtual machines.

Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's personal laptop.

### 8.2.2 Building Blocks of Containers

From an engineering team's point of view, a container is a standard unit of software delivery that facilitates software production and deployment. In order to get a firmer grasp of their capabilities and limitations, we must examine the mechanisms that work as building blocks for containers behind-the-scenes, and enable us to reap their benefits.

#### 8.2.2.1 cgroups

Control groups (cgroups) essentially are a mechanism to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner ([16]).

Using cgroups, the administrator of the system can assign a set of limits to the resource usage of a collection of processes which are bound by the same criteria. The

organization of the groups can be hierarchical, in the sense that every group inherits its configuration from its parent.

The Linux kernel exposes a variety of controllers (subsystems) through the cgroup interface that are used to limit the resource usage of these groups. As an example, the memory controller limits memory usage and `cpuacct` accounts CPU usage.

### 8.2.2.2 Namespaces

Namespaces are Linux kernel feature that enables the partitioning of the kernel resources in a way that different sets of processes have access to different sets of resources. Some examples of these resources are process IDs, file names and files related to network access.

### 8.2.3 Containers are not VMs

While both containers and virtual machines essentially are virtualization implementations, they differ significantly.

One major characteristic of VMs is that they run a completely separate guest OS and emulate its hardware devices, meaning that essentially they virtualize the hardware stack. For example, different VMs, that run on the same host OS, have their own, separate image on disk. VMs also provide strict security and isolation between workloads.

Containers, on the other hand, share the underlying OS. Instead of virtualizing the hardware stack, they virtualize at the operating system level, with multiple containers running on top of the OS kernel directly. This means that containers are far more lightweight. They share the OS kernel, start much faster, and use a fraction of the memory of the machine, compared to booting an entire OS, as in the case of virtual machines.

Containers are bundled only with a minimal set of necessary dependencies (libraries, misc files) and container images usually are (this depends on the containerized application) orders of magnitude smaller than VM images.

### 8.2.4 Docker

Docker is a set of platform-as-a-service products that use operating-system-level virtualization to deliver software in packages. These packages are essentially *containers*, or more specifically: container images, the meaning of which will be explained right below. Docker uses the building blocks mentioned above to create an interface that makes it easier to manipulate and parameterize containers, as well as the applications that run inside them.

#### 8.2.4.1 Images

Containers, as a technology, emerged to cover the need for reusability and reproducibility of software applications. For that to happen, software engineering teams needed a way to freeze the state of containers, so that they can later ship these frozen containers

to users that will run the containerized applications. This frozen version of a container is called a container image, and it's a concept first introduced by Docker.

A container image essentially is a static representation that determines the execution of a container. This means that it contains information about both the structure of the containerized filesystem and which processes will run inside the container. In a few words, a container image is an immutable file which essentially describes a snapshot of the container.

The image's file system is created by stacking up a list of read-only layers, by using a union filesystem. Then, when a container is instantiated from this image, a thin writable layer is added on top of the read-only ones. This layer is also called the "container layer". All changes made to the running container, such as writing new files, modifying existing files and deleting files, are written on this thin writable container layer.

#### 8.2.4.2 Registries

Since images are essentially container specification files, they can be versioned, uploaded and shared to users. These central places where images would be uploaded and hosted are called registries. Specifically Docker has implemented its own registry: DockerHub ([17]). Registries, conceptually, work very similarly to GitHub repositories, with the only exception that they work specifically for images, not just any type of code. Users can upload their images, version them and even have different branches for their images, just like GitHub.

### 8.3 Kubernetes

Containers provide a way for applications to run inside isolated, immutable and reproducible environments. Launching a container virtually what every developer does on a regular basis. The logistical problem arises when the number of applications (and users) grows significantly. In that case, managing a significant number of physical nodes that run user containers, executing health checks on them and ensuring that containers recover from failure is no trivial task.

**Kubernetes** [7] satisfies this need, in addition to providing ways to scale apps dynamically and ways for different containers to communicate with each other and share underlying storage. It is a managing platform for containerized workloads, and is ubiquitous in today's cloud computing landscape.

#### 8.3.1 Overview

The main philosophy behind Kubernetes is that you can declaratively define the desired state for the system, and the system will be constantly monitoring itself and strive to achieve this state. The state is expressed as a set of *YAML* [18] objects that are persisted in a distributed, high-availability key-value store, called *etcd* [19].



### 8.3.2 Controllers and Objects

Kubernetes contains a number of abstractions that represent the state of the system. These abstractions are represented by objects in the Kubernetes API. A Kubernetes object is a “record of intent”. Once the user creates an object, the Kubernetes system will constantly work to ensure that object exists and has the desired status.

Every object in Kubernetes will have some of the following fields:

- **Kind:** The kind of the object. Objects can be, for example, of kind: Pod, Deployment, Service and others.
- **apiVersion:** Specifies the version of the object.
- **Metadata:** Data that helps to uniquely identify the object, including a name string, ID and optional namespace.
- **Spec and Status:** Every Kubernetes object includes two nested object fields that govern the object’s configuration: the *spec* and the *status*. **The spec, which the user provides, describes the desired state and the status describes the actual state of the Object.** At any given time, the Kubernetes Control Plane actively manages an object’s actual state to match the desired state that the user specified.

### 8.3.3 Storage

#### 8.3.3.1 Volumes

A *Pod* that uses volumes specifies in its *spec* which volumes it intends to use, as well as the path that these volumes will be mounted in the containers’ file-systems. Processes that run inside a container “see” a file-system composed from their Docker image and their mounted volumes. The Docker image is at the root of the file-system hierarchy, and any volumes are mounted at the specified paths within the image.

#### 8.3.3.2 *PersistentVolumes* and *PersistentVolumeClaims*

Kubernetes provides the *PersistentVolume* subsystem via its API to users in order to abstract the details of how storage is provided from how it is consumed. For this it provides the *PersistentVolume* and *PersistentVolumeClaim* API Objects.

A *PersistentVolume* is an entity that represents a piece of storage in the cluster that has been provisioned, either statically from a user or dynamically. It is a resource in the cluster just like a *node* is a cluster resource.

A *PersistentVolumeClaim* is a request from a user to consume storage. Just like a *Pod* requests to consume a Node resource, a *PersistentVolumeClaim* requests to consume *PersistentVolume* resource.

## 8.4 Kubeflow

### 8.4.1 Overview

Kubeflow [3] is a machine learning toolkit for Kubernetes that aims at simplifying scaling and deploying ML models to production, by leveraging the advantages of Kubernetes.

Kubeflow started as an open sourcing of the way Google ran *TensorFlow* [20] internally, based on a pipeline called TensorFlow Extended [21]. It began as just a simpler way to run *TensorFlow* jobs on Kubernetes, but has since expanded to be a multi-architecture, multi-cloud framework for running end-to-end machine learning workflows.

Kubeflow is a set of *CustomResourceDefinitions* and Web Applications for manipulating these as well as the Central Dashboard which links them all together to provide a cohesive experience. This includes the Jupyter Notebooks and Pipelines components which we will expose in the next sub-sections. The user is expected to interact with Kubeflow via these UIs.

In the following sub-sections, we will demonstrate Kubeflow's core components.

### 8.4.2 Jupyter Notebooks

This component is responsible for allowing the user to deploy and manipulate Jupyter Notebooks in their Kubeflow cluster. To achieve this, it provides a user friendly UI that enables the user to handle the life-cycle of Notebook *CustomResources*.

### 8.4.3 Pipelines

Kubeflow Pipelines (KFP) [10] is a platform for building and deploying portable, scalable machine learning workflows based on Docker containers and is one of the Kubeflow core components. It's automatically deployed during Kubeflow deployment. The Kubeflow Pipelines platform consists of a user interface (UI) for managing and tracking experiments, jobs, and runs, along with an engine for scheduling multi-step ML workflows. It also comes with an SDK for defining and manipulating pipelines and components. Besides that, there are notebooks for interacting with the system using the SDK.

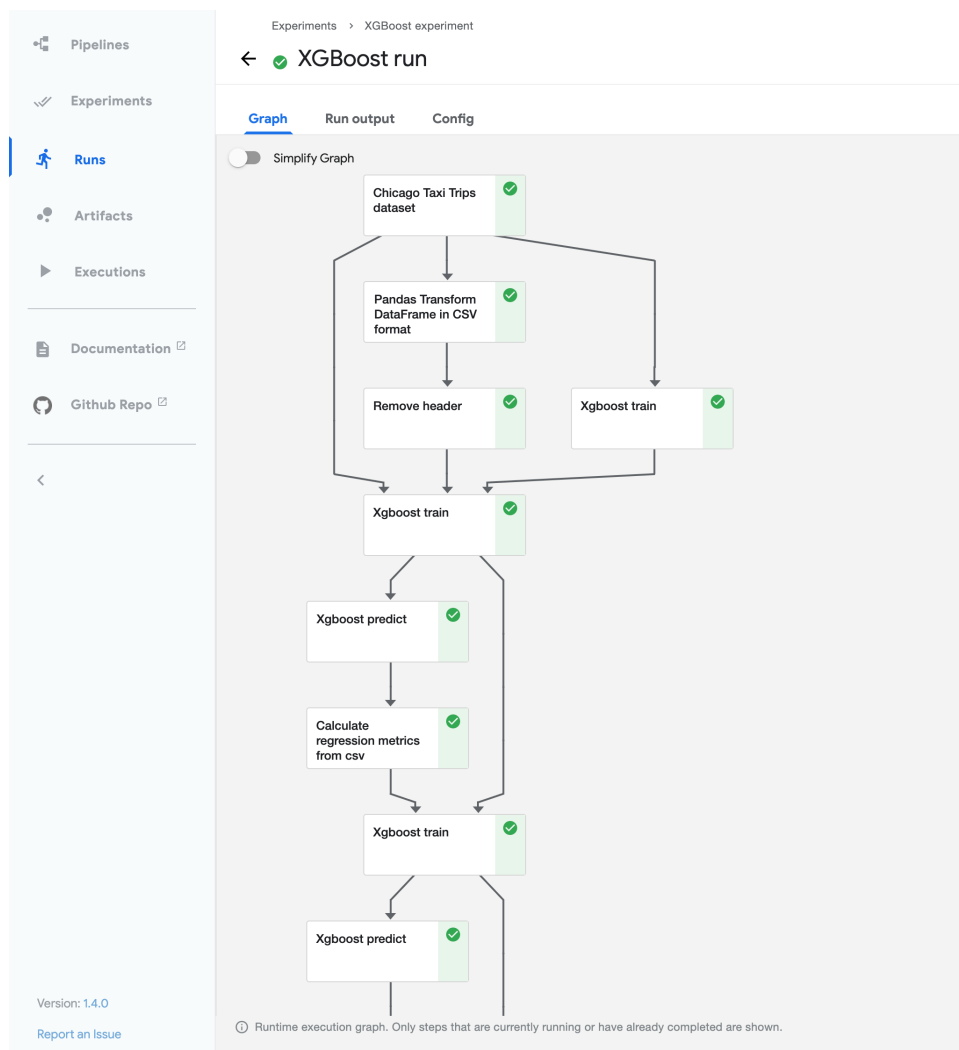


Figure 8.1: The runtime execution graph of a pipeline in the KFP UI

KFP provides end-to-end orchestration, enabling and simplifying the orchestration of machine learning pipelines. On top of that, it is easy for users to try numerous ideas and techniques and manage various trials/experiments. In addition, it enables re-using components and pipelines to quickly create end-to-end solutions without having to rebuild each time.

#### 8.4.3.1 Argo

Argo [22] is a workflow engine. It is an extension to the Kubernetes cluster that makes the execution of workflows possible. The user submits a *YAML* definition of a workflow (*Workflow CustomResourceDefinition*) and then Argo is responsible for initiating tasks in proper order and waiting for them to complete. It also provides a Command Line Interface (CLI) tool as well as a basic User Interface (UI) for the virtual representation of the workflows.

**Kubeflow Pipelines uses Argo as their workflow engine.** The Software Development Kit (SDK) compiles the user's source code into an Argo *Workflow CustomResourceDefinition* which then must be applied to the cluster.

### 8.4.4 Katib

Katib [4] is Kubeflow's component for hyperparameter optimization of models. The main idea behind Katib is to define the training procedure inside a container and run this logic multiple times in order to reach a satisfying set of hyperparameters. This is achieved by creating an *Experiment CustomResource* and by requiring the code to be containerized in a specific way.

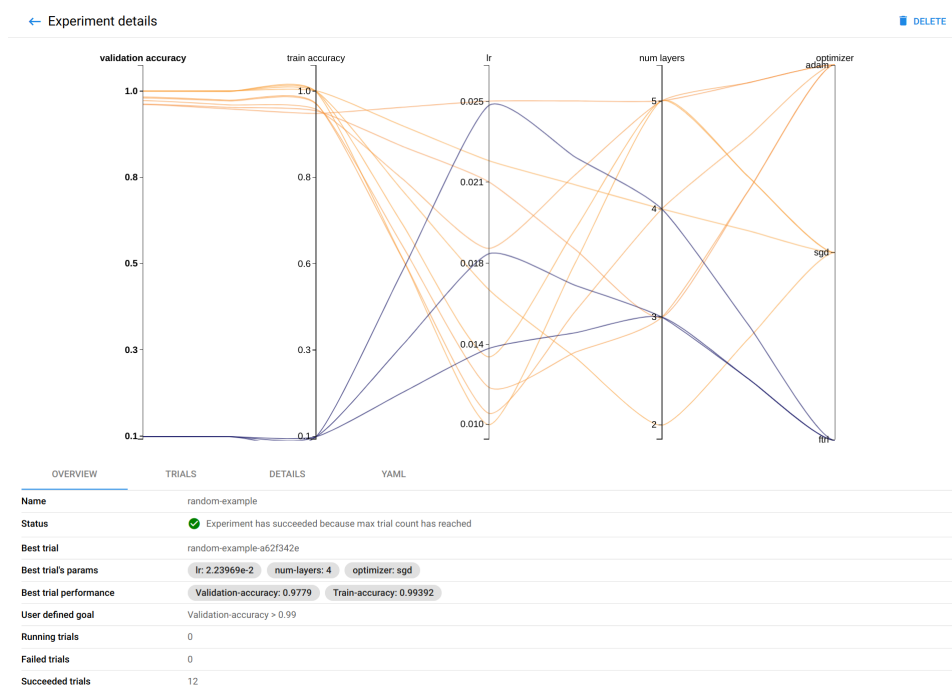


Figure 8.2: Example graph from the Katib UI, showing the level of validation and train accuracy for various combinations of hyperparameter values

The containerized code will need to be able to run standalone. This means that the created container will need to be able to both access the data and train the model. The input in the code will be the hyper-parameter values, either in the *args* field or as environment variables, and will output in a result metric.

The containers structured in this manner can be run multiple times, even in parallel, with different hyper-parameter values as inputs, **in search of a set of them that will accomplish a specified performance quota**. There are also multiple searching strategies for navigating through the hyper-parameters space such as: Cross-Validation, Random search, Bayesian optimization to name a few.

The results from each run are also stored in a central database that is persisted with the use of *PersistentVolumes*. This is all orchestrated from the *Experiment CustomResource Controller* that is responsible for deploying Kubernetes Jobs for training the model, keeping track of the performance of each run, applying the searching algorithm and finally deciding when to stop the optimization process.

### 8.4.5 MiniKF

MiniKF [23] is a single-node instance of KubeFlow, that can be deployed locally or in the cloud. It combines KubeFlow with the Rok Data Management platform which we will describe in the next sub-section.

#### 8.4.5.1 Rok

Rok [24] is a data management and storage platform that allows users to snapshot, version, package, distribute, and clone their full environment along with its data. It is natively integrated with Kubernetes as one of its supported platforms.

It is important to note that in our work for a distributed AutoML process in Kubernetes, we leverage Rok's functionality to take volume snapshots at each step of the process, making the intermediate and final results of our experiments fully reproducible.

## 8.5 Kale and the Kale-SDK

### 8.5.1 Overview

Kale ([11]) stands for "KubeFlow Automated pipeLines Engine" and it is a project that aims at simplifying the Data Science experience of deploying KubeFlow Pipelines workflows. By extending the Jupyter UI, it allows users to deploy Jupyter Notebooks, that are running locally or in the cloud, to KubeFlow Pipelines. This can happen by annotating code cells and clicking a deployment button in the extended Jupyter UI. Kale is responsible for converting the user's annotated Notebook to a working KubeFlow Pipeline, as well as taking care of the data-passing between steps and managing the KubeFlow Pipeline's life-cycle.



Figure 8.3: *The Kale Logo*

Apart from the Jupyter UI extension that we described above, Kale also provides a software development kit, which we will refer to as the **Kale SDK** from this point and on. The Kale SDK allows users to write Python, function-based code and convert it to fully reproducible KubeFlow pipelines without making any change to the original source code. For the purpose of this thesis, we extended the Kale SDK to be able to create AutoML experiments on KubeFlow. Let us now provide an insight on some of the basic concepts of the Kale-SDK that are essential to the understanding of our work.

For a short and thorough overview of Kale's history and basic features refer to the excellent blog post from *Kale's original author*, Stefano Fioravanzo [25]. You can also refer to the official documentation of the Kale-SDK [26]

### 8.5.2 The Step Class

The *Step* class lives under Kale's *step* module, and it enables users to declare Python functions as Kale steps. A Kale step essentially is a callable object that wraps a user-defined function with Kale's marshaling logic (8.5.7).

Users can instantiate a Step object by using the `@step` decorator which can be imported from Kale's *sdk* module. When wrapped around a Python function, the `@step` decorator returns a *Step* object that has its `do_run` attribute overridden by the user-defined Python function. The following code snippet shows such an example.

Listing 8.1: Example of step function decorated with the `@step` decorator

```
1 @step(name="my_step")
2 def step_1(in_1, in_2):
3     # implement the step's business logic here
```

Another way to define a Kale step is through sub-classing the *Step* class and by implementing a `do_run` method. Here is a code example.

Listing 8.2: Example of Kale step that sub-classes the *Step* class

```
1 class ExampleStep(Step):
2     name = "example-step"
3     def do_run(self, param1, param2):
4         # implement the step's business logic here
```

For the purpose of this thesis, we defined a number of custom steps using the sub-classing technique that we described above.

Input parameters are detected automatically by parsing the step's `do_run` function-attribute.

### 8.5.3 The Pipeline Class

The *Pipeline* class lives under Kale's *pipeline* module and is used to define a Kale pipeline, its steps and all their dependencies. It extends *networkx*'s *DiGraph* class ([27]) for directed graphs with self-loops to exploit its underlying graph-related algorithms but also provides helper functions to work with Kale *Step* objects instead of standard *networkx* "nodes". This makes it simpler to access the steps of the pipeline and their attributes.

A Kale Pipeline object can be converted into a KubeFlow pipeline using the Kale Compiler class (8.5.6).

Users of the Kale-SDK are not expected to use *Pipeline* objects directly. Instead, the *api* module of the Kale-SDK provides a `@pipeline` decorator that enables users to easily declare a Kale Pipeline by "wrapping" the decorator around a pipeline function.

Listing 8.3: Example of pipeline function decorated with the `@pipeline` decorator

```
1 @pipeline(name="my_pipeline", experiment="test")
2 def pipeline(param1="dont"):
3     res1 = step_1(param)
```

```

4     if param == "do":
5         res2 = step_2(res1)

```

### 8.5.4 Kale's Domain Specific Language

Kale-SDK's domain specific language is meant to provide a Python-like API to define a pipeline. From this point and on, we will use the acronym: "DSL", to refer to the term: "domain specific language".

The whole essence of Kale's DSL is to allow writing Python functions that describe the architecture of a pipeline and that, without a `@pipeline` decorator, can be run as-is, locally. The only operation required to turn it into a `Pipeline` object should be applying the `@pipeline` decorator.

The DSL currently allows the following Python statements:

1. Function calls with input parameters that are either pipeline parameters or other function call outputs. These function calls correspond to steps of the pipeline.

Listing 8.4: Example of step with input parameter that is a pipeline parameter

```

1     def dsl(param="Hello"):
2         step_1(param)
3         step_2()

```

2. Assignments from (step) function calls. The assigned values are the step outputs. Multiple outputs can be retrieved with tuple assignments.

Listing 8.5: Example of step with input parameter that is another step's output

```

1     def dsl(param="Hello"):
2         my_out = step_1(param)
3         step_2(my_out)

```

3. If-statements with boolean conditions. These conditions must have just one comparison between pipeline parameters and constant values.

Listing 8.6: Example of steps inside if-statements

```

1     def dsl(param1="no", param2="no"):
2         res1 = step_1(param)
3         if param == "yes":
4             res2 = step_2(res1)
5         if param2 == "yes":
6             step_3(res2)

```

### 8.5.5 The `PythonProcessor` Class

The `PythonProcessor` is an API class that lives under Kale's `processors` module and its purpose is to validate a Python function, written in Kale's DSL and convert it into a `Pipeline` object. The constructor of the `PythonProcessor` mainly needs two input arguments:

- **pipeline\_function (Callable)**: A pipeline function, written in Kale DSL (8.5.4). This function essentially describes the entire architecture of the pipeline, that is the flow of data from the first to the last step of the pipeline.
- **config (pipeline.PipelineConfig)**: This is a configuration object, that lives under Kale's *pipeline* module used to store pipeline metadata, such as the name of the pipeline, the name of the KubeFlow pipeline experiment, a description for the pipeline and other such metadata.

The validation of the input pipeline function occurs **during the initialization of the *PythonProcessor* object**, whereas the ***run*** method of the object is responsible for converting the input pipeline function into a *Pipeline* object (8.5.3).

### 8.5.6 The *Compiler* Class

The *Compiler* internal class, converts a Kale *Pipeline* object into a KFP Pipeline. When one uses Kale to run a *@pipeline* decorated function, Kale first creates a Pipeline object, via the *PythonProcessor* that we described in sub-section 8.5.5, and then uses a *Compiler* object to convert it to a KFP pipeline.

The process of converting a Kale Pipeline object to a KubeFlow pipeline is implemented by *Compiler's compile\_and\_run* method, which:

1. Compiles the *Pipeline* object to a *Workflow YAML*.
2. Creates a KubeFlow pipeline by uploading the *Workflow* on KFP.

### 8.5.7 Kale's Data-Passing Mechanism

In order to pass data between steps, Kale automatically provisions a new *PersistentVolume* or uses an existing workspace volume that gets attached to each step container of a pipeline.

Kale injects code at the end of the execution of a step's *do\_run* function to marshal the output objects of the step into this shared *PersistentVolumeClaim* during execution. Similarly, it injects code at the start of the execution of a step, in order to unmarshal the output objects of previous steps and give them as inputs to the current step.

For this purpose, Kale uses its *marshal* module, and more specifically the ***save*** and ***load*** methods of the module that are used to marshal and unmarshal data respectively.

### 8.5.8 Kale's Data Versioning and Snapshots

In the case that Kale runs in a Notebook Server inside a MiniKF instance (8.4.5), it uses the Rok client (8.4.5.1) to:

1. Identify existing workspace/data volumes in the Notebook Server, snapshot them and mount them into the pipeline steps. In this way the workspace of the user (that may contain data-files or installed dependencies) is preserved in the running pipeline.



2. Snapshot volumes at the end of the pipeline run, providing a convenient way to retrieve marshalled objects that were produced during the pipeline's execution, **such as fully-trained model objects or processed data**.
3. Snapshot volumes at the beginning of each step run, providing a convenient way to recover the state of data before a potential step failure.

## 8.6 Machine Learning Meta-Data

Machine Learning Meta-Data (MLMD) [28] is a NoSQL database where we record and retrieve metadata associated with ML workflows and experiments. From this point onwards, we will refer to the Machine Learning Meta-Data store as "MLMD".

MLMD defines entities, that can be stored and retrieved to/from the database. In our work, we associate each one of these entities with a specific concept of a ML workflow or experiment. The main entities that we use throughout our work are:

- **Contexts:** A *Context* is an entity that's meant to encapsulate a group of other entities that all define a "context". We map these entities to KFP Runs.
- **Executions:** An *Execution* corresponds to something that executes/runs. An *Execution* can be part of one (or more) *Contexts*. We map these entities to KFP steps.
- **Artifacts:** An *Artifact* is something that gets consumed or produced by an *Execution*. Additionally, it can be part of one (or more) *Contexts*. In our work we produce *Artifacts* that correspond to a number of different ML-related entities.
- **Attributions:** An *Attribution* declares that an *Artifact* is part of a *Context* entity.
- **Associations:** An *Association* declares that an *Execution* is part of a *Context* entity.
- **Events:** An *Event* declares that an *Artifact* is an input or an output of an *Execution* entity.

Refer to TensorFlow's MLMD user-guides [29] for detailed explanation of the concepts behind these entities.

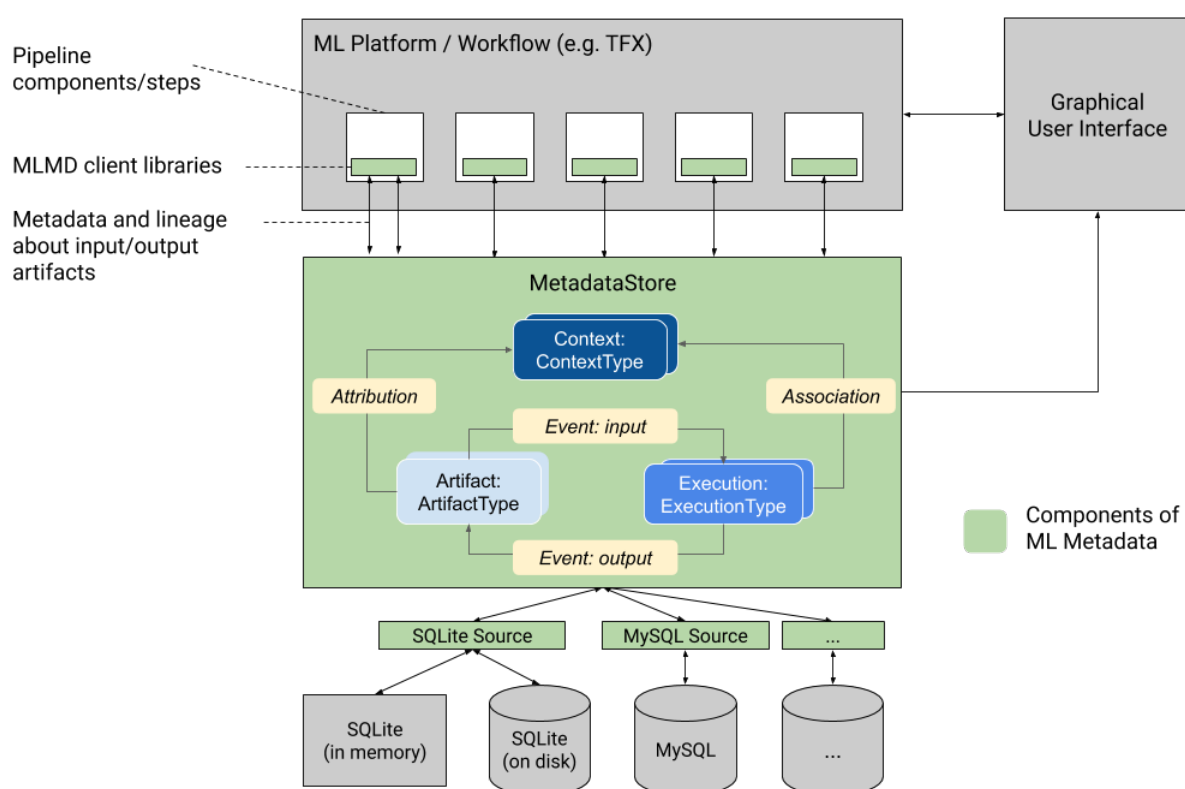


Figure 8.4: An Overview of MLMD

## Our Study of *Auto-sklearn*'s Meta-Learning Mechanism

---

### 9.1 Overview

The *auto-sklearn* ([1]) package is an automated machine learning toolkit that frees a machine learning user from algorithm selection and hyper-parameter tuning. By leveraging recent advantages in Bayesian optimization, meta-learning and ensemble construction, it manages to completely replace any *scikit-learn* ([2]) estimator, for supervised machine learning tasks.

For the purpose of this thesis, we isolated and used *auto-sklearn*'s meta-learning mechanism. So, in the following sections, we will provide a summary of our study on how the mechanism works and a detailed description for the individual parts that make up the mechanism.

Below, we will expose a numbered list of steps that describes this meta-learning mechanism. Essentially, *auto-sklearn*:

1. **extracts a set of meta-feature values** from an input dataset.
2. **compares this meta-feature vector with a number of other meta-feature vectors** that are stored in its meta-learning database and it finds the most similar one. Essentially, each of these other meta-feature vectors corresponds to a specific dataset, so in other words *auto-sklearn* finds the most similar dataset that exists in its meta-learning database.
3. **suggests a set of machine learning pipeline configurations** that scored highly on the most similar dataset. *auto-sklearn* keeps these configurations in its meta-base as well, along with their respective metric scores for each dataset. These suggested configurations are bound to work well on the input dataset.

### 9.2 Machine Learning Pipeline Configurations

#### 9.2.1 Overview

A machine learning pipeline configuration is essentially a detailed description of an entire machine learning pipeline. The *auto-sklearn* package comes with a database (meta-

base) that contains a plethora of such configurations. We will describe this meta-learning database in a following section.

*auto-sklearn*'s goal is to suggest configurations that are bound to score highly for a given machine learning dataset and task. The snippet below shows an example meta-learning configuration for a classification task:

Listing 9.1: A meta-learning configuration for classification

```

1 balancing:strategy, Value: 'weighting'
2 classifier:__choice__, Value: 'libsvm_svc'
3 classifier:libsvm_svc:C, Value: 6384.641073379224
4 classifier:libsvm_svc:coef0, Value: -0.1592835134753816
5 classifier:libsvm_svc:degree, Value: 2
6 classifier:libsvm_svc:gamma, Value: 0.6866143858851854
7 classifier:libsvm_svc:kernel, Value: 'poly'
8 classifier:libsvm_svc:max_iter, Constant: -1
9 classifier:libsvm_svc:shrinking, Value: 'False'
10 classifier:libsvm_svc:tol, Value: 2.6500330000385803e-05
11 data_preprocessing:categorical_transformer:categorical_encoding:__choice__, Value: 'no_encoding'
12 data_preprocessing:categorical_transformer:category_coalescence:__choice__, Value: '
    no_coalescence'
13 data_preprocessing:numerical_transformer:imputation:strategy, Value: 'median'
14 data_preprocessing:numerical_transformer:rescaling:__choice__, Value: 'normalize'
15 feature_preprocessor:__choice__, Value: 'no_preprocessing'

```

Similarly, the following snippet shows a real meta-learning configuration that could be used for a regression task:

Listing 9.2: A meta-learning configuration for regression

```

1 data_preprocessing:categorical_transformer:categorical_encoding:__choice__, Value: 'no_encoding'
2 data_preprocessing:categorical_transformer:category_coalescence:__choice__, Value: '
    no_coalescence'
3 data_preprocessing:numerical_transformer:imputation:strategy, Value: 'most_frequent'
4 data_preprocessing:numerical_transformer:rescaling:__choice__, Value: 'robust_scaler'
5 data_preprocessing:numerical_transformer:rescaling:robust_scaler:q_max, Value:
    0.8280417820125114
6 data_preprocessing:numerical_transformer:rescaling:robust_scaler:q_min, Value:
    0.0781634653874277
7 feature_preprocessor:__choice__, Value: 'kitchen_sinks'
8 feature_preprocessor:kitchen_sinks:gamma, Value: 8.438432240830361e-05
9 feature_preprocessor:kitchen_sinks:n_components, Value: 2984
10 regressor:__choice__, Value: 'ard_regression'
11 regressor:ard_regression:alpha_1, Value: 0.00044036509169446026
12 regressor:ard_regression:alpha_2, Value: 4.039147500668822e-10
13 regressor:ard_regression:fit_intercept, Constant: 'True'
14 regressor:ard_regression:lambda_1, Value: 8.922721154590444e-05
15 regressor:ard_regression:lambda_2, Value: 3.0105431227198885e-05
16 regressor:ard_regression:n_iter, Constant: 300
17 regressor:ard_regression:threshold_lambda, Value: 1899.168836704701

```

```
18 regressor:ard_regression:tol, Value: 0.011611373742389547
```

As it can be seen from above, a configuration is a dictionary-like object that defines **a preprocessing step** and **an estimator step** for a machine learning pipeline. In a following chapter, we will see how we turn these descriptions into actual *sklearn* objects that actually implement preprocessors and estimators.

## 9.2.2 The Configuration Space

The configuration space essentially consists of all the pipeline configurations that are valid for a given ML task. For example, for the case of a *classification* task, the configuration space will contain pipeline configurations that only have *classifiers* as estimators. Similarly, for the case of a *regression* task, the configuration space will contain pipeline configurations that only have *regressors*.

However, the type of the given ML task is not the only criterion that affects the structure of the configuration space. More specifically, the structure of the configuration space is affected by three parameters:

1. **The type of the ML task.**
2. **Whether the input dataset is sparse or not.**
3. **Whether the input dataset contains missing values or not.**

In a following subsection, we will refer to the *XYDataManager* class, which helps us calculate all of the parameters mentioned above. The configuration space is one of the first things that *auto-sklearn*'s meta-learning mechanism calculates, before producing the suggested pipeline configurations.

### 9.2.2.1 The `get_configuration_space` API Function

The *auto-sklearn* package exposes the `get_configuration_space` API function which produces a configuration space object for a given ML task. Here's the signature of the `get_configuration_space` API function:

- **Input Arguments:**

- **info:** A dictionary that contains the information that is necessary to compute the configuration space. This dictionary must contain:
  - \* **task:** An integer that corresponds to the type of the ML task.
  - \* **is\_sparse:** A *bool* value that describes whether the input dataset is sparse or not.
  - \* **has\_missing:** A *bool* value that describes whether there are missing or infinite values in the dataset.
- **include\_estimators:** A list with the names of the estimators to be used, exclusively. This defaults to *None*, so all estimators are included, by default.
- **exclude\_estimators:** A list with the names of the estimators to be excluded. This defaults to *None*, so no estimator is excluded.

- **include\_preprocessors**: A list with the names of the preprocessors to be used, exclusively. This defaults to *None*, so all preprocessors are included, by default.
- **exclude\_preprocessors**: A list with the names of the preprocessors to be excluded. This defaults to *None*, so no preprocessor is excluded, by default.
- **Return Value:**
  - **The configuration space object**

In a following section, we will talk about the *XYDataManager*, a class object provided by *auto-sklearn*, that calculates the **info** dictionary for us.

### 9.2.3 The Meta-Data Directory

*auto-sklearn*'s meta-data directory is an actual directory inside the *auto-sklearn* package where all of *auto-sklearn*'s meta-knowledge is stored. This meta-knowledge is organized in alignment with the *ASlib* format ([30]). This means that meta-data are categorized in sub-directories, depending on three factors:

1. **The metric function**
2. **The type of the task**
3. **The sparsity of the dataset**

So, to give an example, if we imagine a dense dataset for a multiclass classification task, with accuracy as metric, then this would fall under the *accuracy\_multiclass.classification\_dense* sub-directory of *auto-sklearn*'s meta-data directory.

Similarly, for the case of a sparse dataset in a regression task, with mean absolute error as metric, the sub-directory is: *mean\_absolute\_error\_regression\_dense*

### 9.2.4 The XYDataManager Class

The *XYDataManager* class provided by *auto-sklearn* extracts and stores information that is necessary in order to produce suggested configurations for a given ML task. Here are the input arguments of the *XYDataManager* constructor:

- **X**: An array containing the samples of the training dataset.
- **y**: An array containing the targets of the training dataset.
- **X\_test**: An array containing the samples of the testing dataset.
- **y\_test**: An array containing the targets of the testing dataset.
- **task**: An integer that corresponds to the type of the ML task. For the purpose of the Kale-AutoML process, this integer will be either 1, 2 or 4 which corresponds to *regression*, *binary classification* and *multiclass classification* accordingly.
- **feat\_types**: A list of strings that describe the type of each feature. The list has length equal to the number of features in the dataset. Each element of the list can either be **"categorical"** or **"numerical"**<sup>1</sup>.

- **dataset\_name**: The name of the dataset.

The following code snippet shows the code of the *XYDataManager* constructor function:

Listing 9.3: The *XYDataManager* constructor function

```

1  class XYDataManager(AbstractDataManager):
2
3      def __init__(
4          self,
5          X: np.ndarray,
6          y: np.ndarray,
7          X_test: Optional[np.ndarray],
8          y_test: Optional[np.ndarray],
9          task: int,
10         feat_type: List[str],
11         dataset_name: str
12     ):
13         super(XYDataManager, self).__init__(dataset_name)
14
15         self.info['task'] = task
16         if sparse.issparse(X):
17             self.info['is_sparse'] = 1
18             self.info['has_missing'] = np.all(np.isfinite(X.data))
19         else:
20             self.info['is_sparse'] = 0
21             self.info['has_missing'] = np.all(np.isfinite(X))
22
23         label_num = {
24             REGRESSION: 1,
25             BINARY_CLASSIFICATION: 2,
26             MULTIOUTPUT_REGRESSION: y.shape[-1],
27             MULTICLASS_CLASSIFICATION: len(np.unique(y)),
28             MULTILABEL_CLASSIFICATION: y.shape[-1]
29         }
30
31         self.info['label_num'] = label_num[task]
32
33         self.data['X_train'] = X
34         self.data['Y_train'] = y
35         if X_test is not None:
36             self.data['X_test'] = X_test
37         if y_test is not None:
38             self.data['Y_test'] = y_test
39

```

<sup>1</sup>The value of a categorical feature states that a sample belongs to a certain category. These type of features have a finite number of candidate values. Numerical features, on the other hand, express quantitative characteristics of a sample. An example of a categorical feature is a person's country of birth, while a numerical feature is the height of the person.

```

40     if feat_type is not None:
41         for feat in feat_type:
42             allowed_types = ['numerical', 'categorical']
43             if feat.lower() not in allowed_types:
44                 raise ValueError("Entry '%s' in feat_type not in %s" %
45                                 (feat.lower(), str(allowed_types)))
46
47     self.feat_type = feat_type
48
49     # TODO: try to guess task type!
50
51     if len(y.shape) > 2:
52         raise ValueError('y must not have more than two dimensions, '
53                         'but has %d.' % len(y.shape))
54
55     if X.shape[0] != y.shape[0]:
56         raise ValueError('X and y must have the same number of '
57                         'datapoints, but have %d and %d.' % (X.shape[0],
58                                                             y.shape[0]))
59
60     if self.feat_type is None:
61         self.feat_type = ['Numerical'] * X.shape[1]
62     if X.shape[1] != len(self.feat_type):
63         raise ValueError('X and feat_type must have the same number of columns, '
64                         'but are %d and %d.' %
65                         (X.shape[1], len(self.feat_type)))

```

During its instantiation, a *XYDataManager* object extracts information that is necessary for the meta-learning process and it stores it in its **info** class attribute. This attribute is essentially a Python dictionary that contains the following fields:

- **task**: An integer that corresponds to the type of the ML task.
- **is\_sparse**: A *bool* value that describes whether the input dataset is sparse or not. To decide on this, *XYDataManager* uses the **issparse** function ([31]) from **scipy**'s ([14]) **sparse** module.
- **has\_missing**: A *bool* value that describes whether there are missing or infinite values in the dataset.

All of the fields that we described above are essential to finding:

1. the correct **meta-data directory** for a ML task. In a following section, we will describe how auto-sklearn divides its meta-base into directories and how it finds the correct meta-data directory for a given ML task.
2. the **configuration space** in which auto-sklearn will search for candidate meta-learning configurations. We will describe how auto-sklearn finds the configuration space for a given experiment in a following section.



### 9.2.5 The *SimpleRegressionPipeline* and *SimpleClassificationPipeline* Classes

These two classes implement the classification task. They implement a pipeline, which includes pre-processing steps and one estimator step in the end.

An object of these classes gets instantiated by passing a meta-learning configuration object (9.2) to the constructor of the corresponding class. After that, one can call the:

1. **fit** method of the object to fit the pipeline on training dataset.
2. **predict** method of the object to make predictions on a testing dataset.

## 9.3 Meta-Features

### 9.3.1 Overview

In this section, we will describe the basic concepts of the meta-feature extraction mechanism that *auto-sklearn* ([1]) uses under the hood. This is one of the basic parts of the meta-learning kernel that we used for our distributed AutoML process.

Auto-sklearn divides the set of meta-features that it can calculate into two main categories:

1. **Simple Meta-Features:** These meta-features are computationally cheap, and they do not require any transformations on the dataset in order to be calculated.
2. **1HotEncoded Meta-Features:** These meta-features are more computationally expensive, and they are calculated using the **1HotEncoded** feature matrix of the dataset.

We will analyze these two main categories of meta-features in the following sub-sections.

### 9.3.2 Simple Meta-Features

These meta-features are extracted directly from the input Dataset, without any prior transformations or preprocessing, so they are computationally cheap, in general. Here's the full list of the simple meta-features that *auto-sklearn* can calculate:

- **Number of instances**
- **Logarithmic number of instances**
- **Number of classes**
- **Number of features**
- **Logarithmic number of features**
- **Number of features with missing values**
- **Whether values are missing or not**
- **Number of instances with missing Values**
- **Percentage of instances with missing values**
- **Number of features with missing values**
- **Percentage of features with missing values**

- **Number of missing values**
- **Percentage of missing values**
- **Number of numeric features**
- **Number of categorical features**
- **Ratio of numerical to categorical features**
- **Ratio of categorical to numerical features**
- **Ratio of features to instances**
- **Logarithmic ratio of features to instances**
- **Ratio of instances to features**
- **Logarithmic ratio of instances to features**
- **Number of occurrences of each class**
- **Minimum class probability**
- **Maximum class probability**
- **Mean value of class probability**
- **Standard deviation of class probability**
- **Numbers of symbols<sup>2</sup>**
- **Minimum number of symbols**
- **Maximum number of symbols**
- **Standard deviation of numbers of symbols**
- **Sum of numbers of symbols**
- **Class entropy**

### 9.3.3 1HotEncoded Meta-Features

In order to extract these meta-features, the input Dataset undergoes a transformation. More specifically, a **1HotEncoded** feature matrix is created. These meta-features are actually extracted out of that 1HotEncoded feature matrix, not out of the input Dataset itself. In general, the calculation of these features is computationally expensive, since these features are more data-science oriented than the simple meta-features that we listed above. Here's a full list of the 1HotEncoded meta-features that auto-sklearn can calculate:

- **Skewnesses**
- **Minimum skewness**
- **Maximum skewness**
- **Mean skewness**
- **Standard deviation of skewness**
- **Kurtosises**
- **Minimum kurtosis**

---

<sup>2</sup>The term "symbol" expresses a value that a categorical feature can have.

- **Maximum kurtosis**
- **Mean kurtosis**
- **Standard deviation of kurtosis**

Now that we have expanded our knowledge on the types of meta-features that *auto-sklearn* supports, we are ready to explore how one can use *auto-sklearn* to actually calculate these meta-features from a given input dataset.

### 9.3.4 API Functions for Meta-Feature Extraction

To calculate the meta-features that belong in the two categories that we analyzed above, *auto-sklearn* provides two main API functions:

1. **calculate\_all\_metafeatures\_with\_labels**
2. **calculate\_all\_metafeatures\_encoded\_labels**

Both of these functions take the same parameters as inputs. Let's see these parameters in detail:

- **X**: The samples of the training dataset.
- **y**: The targets of the training dataset.
- **categorical**: A list of Boolean values that has a length equal to the number of features in each sample. If an element in the list is True, then the corresponding feature is considered a categorical feature. Otherwise, it's a numerical feature.
- **dataset\_name**: The name of the dataset.
- **dont\_calculate**: A set of meta-features that should not be calculated for that specific input dataset and ML task.

The *dont\_calculate* parameter that we mentioned above is primarily used to exclude meta-features for the case of regression tasks. More specifically, since regression tasks don't have classes as targets, the following meta-features must be excluded:

- **Number of classes**
- **Number of occurrences of each class**
- **Minimum class probability**
- **Maximum class probability**
- **Mean value of class probability**
- **Standard deviation of class probability**
- **Class entropy**

Both of these functions return a *DatasetMetafeatures* object from *auto-sklearn's metalearning.metafeatures* module. This object essentially holds a dictionary of the calculated meta-features in its *metafeature\_values* attribute.

### 9.3.5 The *MetaBase* Class

The *MetaBase* class object is a container for dataset meta-data (meta-feature values), pipeline configurations and experiment results. It is essentially a wrapper around *auto-sklearn*'s meta-knowledge, that is stored in the meta-data directory that we described previously.

In a *MetaBase* object, *auto-sklearn* stores the meta-features of a dataset, as well as the validation results of various pipeline configurations for that specific ML task.

To construct a *MetaBase* object, we need to provide:

- **a configuration space**
- **a meta-data directory**

In addition, after we initialize a *MetaBase* object we can add a dataset entry using the *add\_dataset* method. This method requires:

- **the name of the dataset**
- **a *DatasetMetafeatures* object** that contains the meta-feature values for the corresponding dataset

Adding an input dataset in the *MetaBase* object is essential to calculating the suggested configurations for that dataset using the *suggest\_via\_metalearning* function, which we'll analyze in a following section.

## 9.4 The *suggest\_via\_metalearning* API Function

The *suggest\_via\_metalearning* function of the *autosklearn.metalearning.mismbo* module is one of the most important API functions in *auto-sklearn*'s meta-learning mechanism, since it is the one that actually produces the suggested pipeline configurations for an input dataset and ML task. Let's take a closer look into the signature of the function:

- **Input Arguments:**
  - **meta\_base:** A *MetaBase* object instantiated with the correct meta-data directory and the configuration space in which *suggest\_via\_metalearning* will search for the suggested configurations.
  - **dataset\_name:** The name of the dataset that the suggested configurations will be used for. The dataset must be added in the *MetaBase* object. This can be done with the *MetaBase.add\_dataset* method that we demonstrated in sub-section [subsection 9.3.5](#).
  - **metric:** The metric function.
  - **task:** An integer that corresponds to the type of the ML task.
  - **sparse:** A *bool* that describes whether the input dataset is sparse or not.
  - **num\_initial\_configurations:** The number of suggested configurations to produce.
- **Return Value:**
  - **configurations:** A list of the suggested configurations.

In a following chapter, we will describe how Kale's ML back-end uses the *suggest\_via\_metalearning* to produce the suggested configurations that will later be turned into actual KFP pipelines.

## 9.5 Utility Functions and Classes

### 9.5.1 The *InputValidator* Class

The *InputValidator* provided by the *autosklearn.data.validation* module is a utility class that can be used to make sure the input dataset complies with *auto-sklearn*'s requirements. The basic API method of the *InputValidator* is called *validate* and its signature is shown below:

- **Input Arguments:**
  - **X**: The samples of a dataset.
  - **y**: The targets of a dataset.
  - **is\_classification**: A bool value that expresses whether the task that the input dataset is going to be used for is classification or not.
  
- **Return Values:**
  - **X**: The validated samples.
  - **y**: The validated targets.

*InputValidator.validate* essentially implements two functionalities:

1. It checks that the number of samples matches the number of targets in the dataset.
  
2. It checks that the dataset consists of numerical data only.
  
3. Depending on the type of the task, expressed by the *is\_classification* input argument, it checks that the targets in the dataset can be used for that task (classification or regression).



## Chapter 10

### Our Approach

---

As we explained in a previous chapter, *AutoML* experiments contain steps that can be parallelized. In a local machine, once *auto-sklearn*'s meta-learning mechanism computes the suggested pipeline configurations, *auto-sklearn* trains the corresponding machine learning pipelines locally, as parallel processes, so that the best scoring one is found and returned to the user. Our goal was to transfer this entire process to *Kubernetes*, taking advantage of *Kale*'s machine-learning pipeline orchestration mechanism, so that we train ML models as parallel KubeFlow pipelines. We created a mechanism that allows running *AutoML* experiments efficiently and in a distributed manner, in *Kubernetes*.

Let's outline the steps of the *Kale* process for *AutoML* experiments:

1. The user provides a **dataset** and the **type of the machine learning task** (classification or regression) as input to the `run_automl()` *Kale*-*AutoML* API function.
2. **Kale spins up a KubeFlow pipeline to orchestrate the whole process.** This pipeline is called the *AutoML Orchestrator* and we will be referring to it by that name for the rest of the chapter.
3. **Kale returns an *AutoMLExperiment* object to the user** (return value of `run_automl()`). This object will essentially be a tool for tracking the status of the entire *AutoML* process.
4. Using *auto-sklearn*'s meta-learning kernel, **the *AutoML Orchestrator* computes a list of suggested pipeline configurations** for the user's input dataset and task. Each of these configurations essentially **describes an entire machine learning pipeline**.
5. **For each suggested pipeline configuration, the *AutoML Orchestrator* spins up a new KubeFlow pipeline.** These pipelines are called *Configuration Runs* and we will be referring to them by that name for the rest of the chapter. Each one of these *Configuration Runs* implements the ML pipeline that the corresponding configuration suggests.
6. **Configuration Runs run in parallel**, preprocessing the dataset, training the model, and producing test scores **while the *AutoML Orchestrator* monitors them**.
7. Once all *Configuration Runs* are finished, **the *AutoML Orchestrator* gathers their scores and selects the best Configuration Run**.

8. **The AutoML Orchestrator creates a Katib experiment to further optimize the trained model** of the best scoring Configuration Run.
9. **Kale saves the trained and optimized model and takes a fully reproducible Rok snapshot (8.4.5.1) of the volume that contains it** so that the user can access it later on.

Kale takes snapshots of volumes, not only at the end, but in each step of the AutoML Orchestrator and the Configuration Runs providing a convenient way to retrieve marshalled trained models that were produced during the AutoML process.

**This thesis mainly focuses on steps 3 to 7 of the Kale-AutoML process.** Nevertheless, we will provide a sufficient analysis of the functionality of the rest of the mechanism as well.



## 10.1 The `run_automl` API Function

In this section, we will describe what the Kale API function for AutoML experiments looks like. Essentially, users will run AutoML experiments in KubeFlow, with just a single Python function call. This function, as we mentioned above, is called: `run_automl` and it is essentially an entry-point to the AutoML mechanism.

Listing 10.1: The `run_automl` API function for creating AutoML experiments with Kale

```

1  def run_automl(
2      dataset: Dataset, task: MLTask, metric: Callable,
3      number_of_configurations: int = 5,
4      max_parallel_configurations: int = 3,
5      tuner: Optional[katib.V1beta1ExperimentSpec] = None
6  ) -> AutoMLExperiment:
7      """Runs an AutoML pipeline to find the best model for the input dataset.
8
9      [... Explain how the AutoML process works ...]
10
11     Args:
12         dataset (common.artifacts.Dataset): The input dataset for the ML task
13         task (types.MLTask): One of kale.ml.Task
14         metric (Callable): A callable object with the following call signature
15
16         >>> class my_metric:
17             >>>     def __call__(self, target, x_test):
18                 >>>         return self._compute_my_metric_value(target, x_test)
19
20         The name of the logged metric will be 'metric.name', if the object
21         has such attribute. Otherwise 'metric.__name__'.
22         (Auto)SKLearn metrics are supported, example:
23
24         >>> from autosklearn.metrics import accuracy
25
26         To log a different metric name from the input function name
27         ('foo.__name__'), do:
28
29         >>> foo.name = "<custom_name>"
30
31         number_of_configurations (int): The N-best configurations to run
32             (defaults to 5)
33         max_parallel_configurations (int): The maximum number of Configuration
34             Runs to run in parallel (defaults to 3)
35         tuner (katib.V1beta1ExperimentSpec): Provide a Katib spec to run HP
36             Tuning over the best performing configuration.
37
38         Cannot set algorithm and parameters. To set objective
39         configuration, don't set metric name:
40

```

```

41         >>> katib.V1beta1ExperimentSpec(
42         >>>     objective=katib.V1beta1ObjectiveSpec(
43         >>>         goal=0.99,
44         >>>         type="maximize")
45
46     Returns: An AutoMLExperiment object to track the state of the experiment.
47     """
48     if tuner:
49         if tuner.algorithm or tuner.parameters:
50             raise ValueError("Tuner: Cannot specify 'algorithm', or"
51                               " 'parameters', during an AutoML experiment")
52         if tuner.objective:
53             if (tuner.objective.objective_metric_name
54                 or tuner.objective.additional_metric_names):
55                 raise ValueError("Cannot specify metric name when running"
56                                   " AutoML experiment")
57
58     pipeline_name = "automl-orchestrate"
59     utils.rm_r(ML_ASSETS_DIR)
60     marshal.set_data_dir(ML_ASSETS_DIR)
61
62     variables = ["dataset", "task", "metric", "number_of_configurations",
63                "max_parallel_configurations"]
64     if tuner:
65         variables.append("tuner")
66     for v in variables:
67         marshal.save(locals()[v], v)
68
69     volumes = rokutils.interactive_snapshot_and_get_volumes()
70     pipeline_config = PipelineConfig(
71         pipeline_name=pipeline_name,
72         experiment_name=_auto_ml_experiment_name(),
73         volumes=volumes)
74     pipeline = PythonProcessor(automl_orchestrate, pipeline_config).run()
75     pipeline.input_pipeline_parameters["hp_tune"] = ("true" if tuner
76                                                    else "false")
77     run = Compiler(pipeline).compile_and_run()
78     return AutoMLExperiment(run.id)

```

Users can import the `run_automl()` function with a single Python import command:

```
» from kale.ml import run_automl
```

Let us give a detailed explanation of the input arguments of `run_automl()`:

- **dataset:** The input dataset for the AutoML Experiment. This must be a Kale Dataset object that contains the entire machine learning dataset of the user. We will describe the Kale Dataset class in a following sub-section (10.1.1).
- **task:** The type of the supervised machine learning task. This can be either:

1. binary classification
  2. multi-class classification
  3. simple regression
- **metric**: A callable metric function that will be used to evaluate the trained model produced by each Configuration Run.
  - **number\_of\_configurations**: The number of suggested configurations that the AutoML Orchestrator will extract. The AutoML Orchestrator will eventually create a Configuration Run for each of these configurations.
  - **max\_parallel\_configurations**: The maximum number of Configuration Runs that will run in parallel. Since the resources of a Kubernetes cluster (e.g. its CPU capacity) are limited, the user must have a say on how much computing power will be consumed during the most demanding part of the AutoML process: the parallel execution of the Configuration Runs.
  - **tuner**: A Katib experiment specification that will be used for further parameter optimization of the best performing model.

As we mentioned in the beginning of this chapter, the `run_automl()` API function **returns an `AutoMLExperiment` object** (10.4.2). This object, and essentially its methods, will enable the user to track the status of the experiment. We will elaborate more on the status tracking of the experiment in section 10.4, after we have first analyzed the functionality behind the **AutoML Orchestrator** pipeline (10.2) and the **Configuration Runs** (10.3).

Here is a **numbered list of steps** that describes how `run_automl()` API function manages to touch off the entire AutoML process:

1. **It uses Kale's marshaling mechanism to store its input parameters in the Pod's volume and then takes a Rok snapshot of the volume** so that the steps of the AutoML Orchestrator and Configuration Runs can mount the cloned volume to find and load the parameters.
2. **It instantiates a `kale.PipelineConfig` object** with all the basic descriptive information (e.g: the name) of the AutoML Orchestrator.
3. **It instantiates a `kale.processors.PythonProcessor` object** (8.5.5) with a Python function written in Kale DSL code (8.5.4). This Python function is called `automl_orchestrate` and it essentially describes the entire ML functionality of the AutoML Orchestrator. We expose the architecture of the `automl_orchestrate` function in sub-section 10.2.1. The `PythonProcessor` (8.5.5) evaluates this pipeline function and returns a Pipeline object (8.5.3).
4. **It compiles the Pipeline object into an Argo Workflow and applies it to Kubernetes.** This happens using the `kale.Compiler` object and its `compile_and_run` method (8.5.6).
5. **It returns an `AutoMLExperiment` object, instantiated with the ID of the newly created AutoML Orchestrator.** This object allows the user to monitor the status of the AutoML Orchestrator and the Configuration Runs and also to view the metric scores of the Configuration Runs, when they are available.

### 10.1.1 The Dataset Object

In order to simplify the `run_automl()` function signature, we decided to represent the user's input dataset and all its components as one abstract entity. This entity is called a Kale Dataset, and here are its attributes:

- **name:** The name of the machine learning dataset.
- **features:** The features of the part of the dataset that will be used for training.
- **targets:** The targets of the part of the dataset that will be used for training.
- **features\_test:** The features of the part of the dataset that will be used for testing.
- **targets\_test:** The targets of the part of the dataset that will be used for testing.

## 10.2 The AutoML Orchestrator Pipeline

In this section, we will describe the pipeline that creates and monitors the entire AutoML experiment. This pipeline is called the **AutoML Orchestrator**, and it essentially consists of six steps that all run Kale code. Here's an overview of the mechanism that each step implements:

1. **get-metalearning-configurations:** Using the auto-sklearn kernel, this step extracts a set of meta-features from the input dataset and produces a list of suggested meta-learning configurations.
2. **run-metalearning-configurations:** This step takes the configurations generated by step 1 and starts a new Configuration Run for each one of them. These Configuration Runs all belong to the same KFP experiment as the AutoML Orchestrator. The maximum number of Configuration Runs that run in parallel is defined by the `max_parallel_configurations` option in `run_automl`.
3. **monitor-kfp-runs:** This step waits for the Configuration Runs started by step 2 to complete.
4. **get-best-configuration:** Every Configuration Run outputs a metric score, which denotes the performance of the corresponding trained model. This step gathers the metric scores from all Configuration Runs and picks the best performing one.
5. **run-katib-experiment:** This step is optional. If the user does not provide a *tuner* object in `run_automl`, then the AutoML Orchestrator completes at step 4. In this step, Kale takes the meta-learning configuration that corresponds to the best Configuration Run from step 4, and creates a Katib experiment to perform hyper-parameter tuning over the model. This optimizes the model by searching the parameter space of its architecture for parameter values that enable it to perform even better.
6. **monitor-katib-experiment:** This step waits for the Katib experiment created in step 5 to complete.

Let us show the following image that showcases an example AutoML Orchestrator pipeline, as shown in the KubeFlow Pipelines UI.

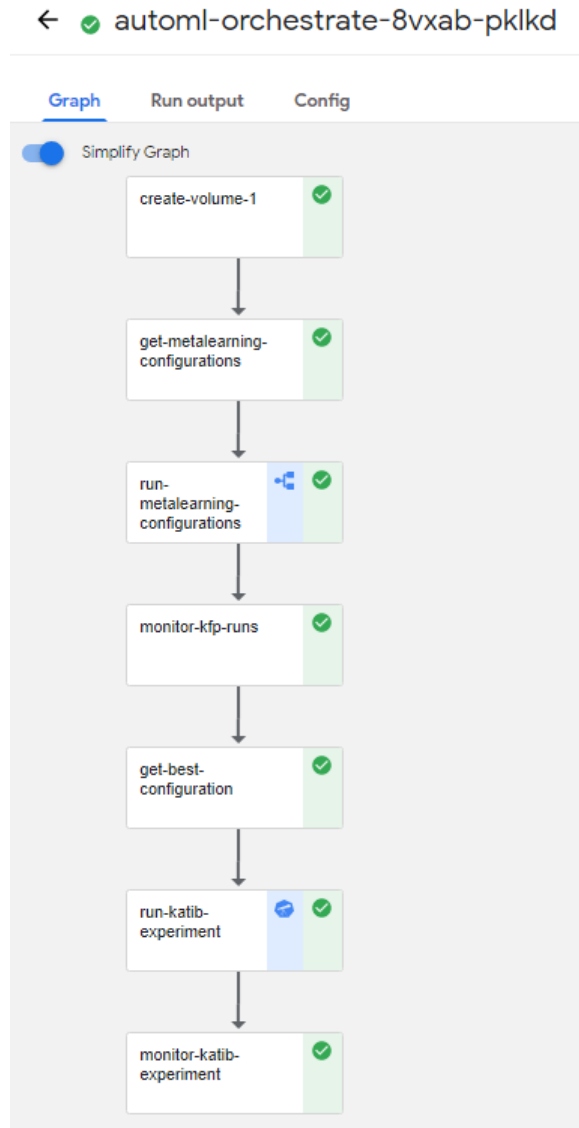


Figure 10.1: Example AutoML Orchestrator pipeline in the KFP UI

### 10.2.1 The `automl_orchestrate` Pipeline Function

The `automl_orchestrate` function is a Python function, written in Kale’s domain-specific language (8.5.4), that essentially describes the structure of the AutoML Orchestrator that the `run_automl` API function creates. Since it’s written in Kale DSL, it uses Kale `Step` (8.5.2) instances to represent steps in the pipeline. Here is the corresponding code snippet:

Listing 10.2: The `automl_orchestrate` pipeline function

```

1 def automl_orchestrate(hp_tune="false"):
2     """Auto ML pipeline."""
3     (configurations,
4      kale_dataset_id) = GetMetaLearningConfigurations()(
5         ml_assets_marshal_path("dataset.dill.pkl"),
6         ml_assets_marshal_path("task.dill.pkl"),

```

```

7     ml_assets_marshall_path("metric.joblib"),
8     ml_assets_marshall_path("number_of_configurations.dill.pkl")
9
10    run_ids = RunMetaLearningConfigurations()(
11        configurations,
12        ml_assets_marshall_path("max_parallel_configurations.dill.pkl"),
13        kale_dataset_id)
14    run_ids = MonitorKFPRuns()(run_ids)
15    best_configuration = GetBestConfiguration()(
16        run_ids,
17        configurations,
18        ml_assets_marshall_path("metric.joblib"))
19    if hp_tune == "true":
20        katib_experiment_name = RunKatibExperiment()(
21            ml_assets_marshall_path("tuner.dill.pkl"),
22            best_configuration,
23            ml_assets_marshall_path("metric.joblib"),
24            kale_dataset_id)
25        MonitorKatibExperiment()(katib_experiment_name)

```

Note that some of the step inputs in the pipeline function are **paths to marshal objects** that correspond to the marshalled input parameters of the *run\_automl* (10.1) API function. As we see from the previous code snippet, the AutoML Orchestrator pipeline that the *automl\_orchestrate* describes, consists of six Kale Steps:

1. **GetMetaLearningConfigurations**
2. **RunMetaLearningConfigurations**
3. **MonitorKFPRuns**
4. **GetBestConfiguration**
5. **RunKatibExperiment**
6. **MonitorKatibExperiment**

Each of these, is a regular Kale step that implements a specific part of the AutoML Orchestrator pipeline. Followingly, we will expose the functionality of each one of these steps.

### 10.2.2 The *GetMetaLearningConfigurations* Step

In this subsection, we will expose and explain the mechanism that the *get-metalearning-configurations* step implements. We will describe how this step uses *auto-sklearn*'s meta-learning kernel to generate a list of candidate pipeline configurations that are likely to perform well for a given ML task. Let's view the class definition of the *get-metalearning-configurations* step and the ML code (*do\_run* method) that gets executed inside the step:

Listing 10.3: The *do\_run()* method of the *get-metalearning-configurations* step

```

1 class GetMetaLearningConfigurations(Step):
2     """Produce ML suggestions from a dataset using AutoSKLearn MetaLearning.

```

```

3
4     Ins:
5         dataset (Dataset):
6         task:
7         metric (Callable):
8         number_of_configurations (int)
9
10    Outs:
11        configurations (List[Configuration])
12        kale_dataset_id (int)
13
14    MLMD Inputs:
15        kale.Dataset
16
17    MLMD Outputs:
18        kale.AutoMLConfiguration (#'number_of_configurations')
19    """
20    name = "get-metalearning-configurations"
21    outs = Param.odict(["configurations", "kale_dataset_id"], step_name=name)
22
23    def do_run(self, dataset, task, metric, number_of_configurations):
24        """Implementations of GetMetaLearningConfigurations."""
25        from kale.ml import metalearning
26
27        dataset_artifact = self._submit_and_link_dataset_artifact(dataset)
28        configurations = metalearning.compute_configs(dataset, task, metric,
29                                                    number_of_configurations)
30        for idx, configuration in enumerate(configurations):
31            self._submit_configuration_artifact(configuration, idx)
32        return configurations, dataset_artifact.id

```

### 10.2.2.1 The `compute_configs` Function

The process of computing the configurations is essentially implemented by a single function of our `kale.ml.metalearning` module, called: **`compute_configs`**. The following list describes the signature of the function:

- **Input Arguments:**

- **dataset:** A Dataset class object that contains the input dataset for the machine learning task.
- **task:** A `kale.types.MLTask` that describes the type of the machine learning task.

Listing 10.4: The `kale.types.MLTask Enum`

```

1     class MLTask(enum.Enum):
2         """Enum class for Machine Learning tasks."""
3
4         BINARY_CLASSIFICATION = 1
5         MULTICLASS_CLASSIFICATION = 2

```

```
6 SIMPLE_REGRESSION = 4
```

- **metric**: The callable metric function to be used in order to evaluate the trained estimator.
- **number\_of\_suggestions**: The number of suggested pipeline configurations to be returned.

- **Return Value:**

- **configurations**: A list of pipeline configurations ([section 9.2](#)).

The procedure of computing the pipeline configurations is a complex one as it requires a large number of intermediate calculations. Here is **a numbered list of steps** that describes the mechanism behind `compute_configs`:

1. **It validates the input dataset** provided by the user. This is done by calling the `_validate_dataset` function from our `kale.ml.metalearning` module which:
  - (a) checks that the number of samples matches the number of targets.
  - (b) checks that the dataset consists of numerical data only.
  - (c) calculates a list of strings that describes whether each feature in the dataset is *categorical* or *numerical*.

We will analyze this function’s mechanism in a following section.

2. It calculates an array of metafeatures ([section 9.3](#)) by using the `calculate_all_metafeatures` function of our `kale.ml.metafeatures` module. We will analyze this function’s mechanism in a following section.
3. It creates a **`XYDataManager`** ([subsection 9.2.4](#)) object, provided by `auto-sklearn`, where it stores the dataset, the type of the ML task and the list of feature types. Essentially, the constructor of the `XYDataManager` extracts and stores useful information about our experiment, such as whether the input dataset is *sparse* or not. This information is essential for `Kale` to calculate the configuration space ([subsection 9.2.2](#)) which is essential for `auto-sklearn` to suggest accurate meta-learning configurations, later on in the process.
4. It finds the metadata directory ([subsection 9.2.3](#)) that corresponds to the combination of:
  - the type of the metric
  - the type of the ML task
  - whether the dataset is sparse or not.

To do this, `compute_configs` calls `find_metadata_dir`, a utility function from `Kale’s ml.utils` module. We will expose and describe this function in a following section.

5. It finds the configuration space ([subsection 9.2.2](#)) in which `auto-sklearn` will search for pipeline configurations. For this purpose, `compute_configs` calls `get_configuration_space` from the `autosklearn.util.pipeline` module.
6. It creates a `MetaBase` object ([subsection 9.3.5](#)) which holds `auto-sklearn`’s entire meta-learning data-base.



7. It calculates and returns the list of suggested pipeline configurations (section 9.2).

Here is the code that runs inside *compute\_configs*:

Listing 10.5: The *compute\_configs* function that produces a list of suggested pipeline configurations

```

1  def compute_configs(dataset: Dataset,
2      task: MLTask,
3      metric: Callable,
4      number_of_configurations: int) -> List[Configuration]:
5      """Use the AutoSKLearn MetaLearning system to produce ML configurations.
6
7      The AutoSKLearn MetaLearning system is based on prior knowledge on how
8      certain Machine Learning models perform on a set of known datasets.
9      AutoSKLearn can use this prior knowledge to suggest some Machine Learning
10     configurations that are supposed to perform well on a new, previously
11     unseen, dataset."""
12     log.info("Getting suggested configurations...")
13
14     task = mltask_to_string(task)
15
16     validated_dataset, feature_types = _validate_dataset(dataset, task)
17     task_type = extract_task_type(y=validated_dataset.targets, task=task)
18
19     metafeatures = calculate_all_metafeatures(x=validated_dataset.features,
20                                             y=validated_dataset.targets,
21                                             dataset_name=dataset.name,
22                                             task_type=task_type,
23                                             feature_types=feature_types)
24
25     # XYDataManager does some validation to the dataset and the list of feature
26     # types. It also finds if the dataset is sparse or not - useful for
27     # detecting the metadata directory.
28     datamanager = XYDataManager(X=validated_dataset.features,
29                                y=validated_dataset.targets,
30                                X_test=validated_dataset.features_test,
31                                y_test=validated_dataset.targets_test,
32                                task=task_type,
33                                feat_type=feature_types,
34                                dataset_name=dataset.name)
35     is_sparse = datamanager.info["is_sparse"]
36
37     metadata_directory = find_metadata_dir(task_type, metric, is_sparse)
38     config_space = get_configuration_space(datamanager.info)
39     # The MetaBase object is a container for metafeatures, configurations
40     # and their respective scores.
41     meta_base = MetaBase(config_space, metadata_directory)
42     meta_base.add_dataset(dataset.name, metafeatures)
43     configurations = suggest_via_metalearning(

```

```

44     meta_base=meta_base, dataset_name=dataset.name,
45     metric=metric, task=task_type, sparse=is_sparse,
46     num_initial_configurations=number_of_configurations)
47     return configurations

```

### 10.2.2.2 The `_validate_dataset` Function

The `_validate_dataset` function, as its name suggests, validates the input dataset provided by the user. It belongs in our `kale.ml.metalearning` module and it essentially uses the `InputValidator` (subsection 9.5.1) class from the `autosklearn.data.validation` module to validate the input dataset. The `_validate_dataset` function has the following signature:

- **Input Arguments:**

- **dataset:** The input dataset.
- **task:** The type of the task. This must be a string argument that is either "classification" or "regression".

- **Return Values:**

- **validated\_dataset:** The validated dataset.
- **feature\_types:** A list of strings that specifies whether each feature is "categorical" or numerical.

The `_validate_dataset` function essentially consists of three parts. More specifically it:

1. checks that the number of samples matches the number of targets.
2. checks that the dataset consists of numerical data only.
3. calculates a list of strings that describes whether each feature in the dataset is *categorical* or *numerical*.

The following snippet shows the code that runs inside the `_validate_dataset` function.

Listing 10.6: The `_validate_dataset` function of the `kale.ml.metalearning` module

```

1  def _validate_dataset(dataset: Dataset,
2                          task: str = "classification") -> (
3                          Tuple[Dataset, List[str]]):
4      """Validate and process the input features and targets.
5
6      Use AutoSKLearn 'InputValidator' to check if the input dataset
7      is valid (e.g: the number of samples matches the number of targets).
8      Also, auto-sklearn does some "polishing" transformations to the dataset.
9
10     During the validation, 'InputValidator' also determines the feature
11     type for all the input features. A feature type can either be "numerical"
12     or "categorical".
13
14     Args:
15     dataset (Dataset): A Dataset class object that contains the input

```

```

16         dataset for the ML task.
17         task (str): The type of the ML task (classification | regression).
18
19     Returns:
20         Dataset, List(str): The validated dataset and a list of feature types
21         for all features (either "numerical" or "categorical").
22     """
23     is_classification = (task == "classification")
24     input_validator = InputValidator()
25     x, y = input_validator.validate(X=dataset.features, y=dataset.targets,
26                                   is_classification=is_classification)
27     x_test, y_test = input_validator.validate(
28         X=dataset.features_test, y=dataset.targets_test,
29         is_classification=is_classification)
30
31     validated_dataset = copy.deepcopy(dataset)
32     validated_dataset.features = x
33     validated_dataset.targets = y
34     validated_dataset.features_test = x_test
35     validated_dataset.targets_test = y_test
36     return validated_dataset, input_validator.feature_types

```

### 10.2.2.3 The *find\_metadata\_dir* Function

The *find\_metadata\_dir* function, as its name suggests, finds the path to the meta-data directory that corresponds to a given ML task. The function belongs in our *kale.ml.utils* module.

As we described in [subsection 9.2.3](#), *auto-sklearn* organizes its meta-data directory with respect to three characteristics of the ML task:

1. **The metric function**
2. **The type of the task**
3. **The sparsity of the dataset**

Subsequently, our *find\_metadata\_dir* function has the following signature:

- **Input Arguments:**

- **task\_type:** A *kale.types.MLTask* that describes the type of the machine learning task.
- **metric:** The metric function that is used to evaluate the results of the experiment.
- **is\_sparse:** A *bool* value that describes whether the input dataset is sparse or not.

- **Return Value:**

- **metadata\_directory:** A path to a meta-data directory inside *auto-sklearn's* installation directory.

Here is a snippet with the code that runs inside `find_metadata_dir`:

Listing 10.7: The `compute_configs` function that finds the path to the meta-data directory that corresponds to a given ML task

```

1 def find_metadata_dir(task_type: int, metric: Callable, is_sparse: int):
2     """Find the directory where auto-sklearn stores its meta-knowledge.
3
4     Note:
5         The directory structure follows the 'Algorithm Selection Library'
6         (ASLib) format. See https://www.automl.org/automated-algorithm-design/algorithm-selection/aslib/ # noqa: 501
7
8     Args:
9         task_type (int): The type of the ML task.
10        metric (callable): The metric that is used for the ML task.
11        is_sparse (int): Whether the dataset is sparse or not.
12
13    Returns:
14        str: path to the auto-sklearn metadata directory.
15
16    Raises:
17        RuntimeError: When the auto-sklearn metadata directory cannot be found.
18    """
19    log.info("Finding metadata dir...")
20    metalearning_directory = os.path.dirname(autosklearn.metalearning.__file__)
21    # The auto-sklearn metadata directory doesn't provide metadata for
22    # multi-label classification. auto-sklearn reverts to using binary
23    # classification as well, so we copy this behaviour.
24    if task_type == constants.MULTILABEL_CLASSIFICATION:
25        meta_task = constants.BINARY_CLASSIFICATION
26    else:
27        meta_task = task_type
28    metalearning_files_dir = "%s_%s_%s" % (
29        metric, constants.TASK_TYPES_TO_STRING[meta_task],
30        "sparse" if is_sparse else "dense")
31    metadata_directory = os.path.join(
32        metalearning_directory, "files", metalearning_files_dir)
33    if not os.path.exists(metadata_directory):
34        raise RuntimeError("Metadata directory %s does not exist."
35                           % metadata_directory)
36    log.info("Metadata directory: %s", metadata_directory)
37    return metadata_directory

```

#### 10.2.2.4 The `_submit_configuration_artifact` Method

In order to sustain a lineage of the entire AutoML experiment, we enforce steps to create and submit Artifacts to the MLMD database. The `_submit_configuration_artifact` method creates an `AutoMLConfiguration` Artifact (subsubsection 10.4.1.2) out of a given

pipeline configuration.

Listing 10.8: The `_submit_configuration_artifact` method of `GetMetaLearningConfigurations` that creates and submits an `AutoMLConfiguration` Artifact for a given pipeline configuration

```

1  def _submit_configuration_artifact(self, configuration, idx):
2      from kale.ml import utils
3      from kale.common.artifacts import AutoMLConfiguration
4
5      mlmd = mlmdutils.get_mlmd_instance()
6      config_summary = utils.get_configuration_summary(configuration)
7      config = AutoMLConfiguration(
8          config_summary=config_summary,
9          run_id=kfputils.format_kfp_run_id_uri(mlmd.run_uuid),
10         estimator_name="Configuration %s: %s"
11             % (idx + 1, config_summary["name"]))
12     config.assign_list_index(idx + 1)
13     config_artifact = config.submit_artifact()
14     mlmd.link_artifact_as_output(config_artifact.id)

```

As we can see, the `AutoMLConfiguration` class expects a `config_summary` input parameter which describes all the important information about the **estimator** (classifier, or regressor) of the pipeline configuration. More specifically, `config_summary` is expected to be a dictionary that describes the estimator with the following fields:

- **name:** The name of the architecture of the estimator.
- **parameters:** A dictionary of name-value mappings for the hyper-parameters of the estimator.

To produce the `config_summary` parameter, this step uses the `get_configuration_summary` utility function from Kale's `ml.utils` module:

Listing 10.9: The `get_configuration_summary` utility function that creates a summary dictionary out of a given pipeline configuration

```

1  def get_configuration_summary(configuration: Configuration) -> Dict[str, Any]:
2      """Return an opinionated summary of the input configuration.
3
4      The output of this function can be used to pretty-print a configuration,
5      with just the right information, or to upload the configuration to the
6      artifact store.
7
8      Args:
9          configuration: A suggested configuration extracted by auto-sklearn.
10
11     Returns:
12         dict: A dictionary that describes the learner (classifier, or
13             regressor) of the configuration, with the following fields:
14
15         * 'name': The name of the model

```

```

16         * 'parameters': A dictionary of hyperparameters.
17
18     Raises:
19         ValueError: If cannot find a supported learner type. Supported
20             learner types are 'classifier:__choice__' and
21             'regressor:__choice__'.
22     """
23     if configuration.get("classifier:__choice__"):
24         name = _get_classifier_name(configuration)
25         params = _get_params(configuration, "classifier")
26     elif configuration.get("regressor:__choice__"):
27         name = _get_regressor_name(configuration)
28         params = _get_params(configuration, "regressor")
29     else:
30         raise ValueError("Could not find a model in the input configuration.")
31
32     return {"name": name, "parameters": params}

```

### 10.2.3 The *RunMetaLearningConfigurations* Step

In this subsection, we will expose and explain the mechanism that the *RunMetaLearningConfigurations* step implements. In summary, this step takes as input the list of meta-learning configurations that were produced by the previous step, and for each one of them, it spins up an new Configuration Run which implements the ML pipeline that the corresponding configuration suggests. Let's view the ML code that runs inside the Kale step, starting from its class definition and the *do\_run* method:

Listing 10.10: The class definition and *do\_run()* method of the *run-metalearning-configurations* step

```

1 class RunMetaLearningConfigurations(Step):
2     """Run MetaLearning suggestions as KFP pipelines.
3
4     Ins:
5         configurations (List[Configuration])
6         max_parallel_configurations (int)
7         kale_dataset_id (int)
8
9     Outs:
10        run_ids (List[str])
11    """
12    name = "run-metalearning-configurations"
13    outs = Param.odict(["run_ids"], step_name=name)
14    actions = ["RunKFPPipelines"]
15
16    def do_run(self, configurations, max_parallel_configurations,
17              kale_dataset_id):
18        """Implementation of RunMetaLearningConfigurations."""

```

```

19     from time import sleep
20     from kale import marshal
21     from kale.common import mlmdutils
22     from kale.ml.utils import ML_ASSETS_DIR
23     from kale.common.artifacts import AutoMLConfiguration
24
25     self.vars["run_ids"] = []
26     marshal.set_data_dir(ML_ASSETS_DIR)
27
28     mlmd = mlmdutils.get_mlmd_instance()
29     # Get all Artifacts that are attributed to the Context of the AutoML
30     # Orchestrator. The SKLearnTransformer step of each new Configuration
31     # Run should link the corresponding AutoMLConfiguration Artifact as its
32     # input. So, we should pass the corresponding AutoMLConfiguration
33     # Artifact ID to each Configuration Run.
34     kale_config_artifacts = mlmdutils.get_artifacts_by_context_and_type(
35         context_id=mlmd.run_context.id,
36         type_name=AutoMLConfiguration.artifact_type_name,
37         sorted=True)
38     if len(kale_config_artifacts) != len(configurations):
39         raise RuntimeError("Founds %d MLMD configuration artifacts but"
40                             " %d configuration were provided as input."
41                             % (len(kale_config_artifacts),
42                                len(configurations)))
43
44     # Start all configurations with a reconciliation loop to avoid having
45     # more than max_parallel_configurations running concurrently
46     while configurations:
47         if self._running_ids() >= max_parallel_configurations:
48             log.info("Cannot start a new configuration. Max parallel"
49                     " configurations cap is set to %d. Waiting for a"
50                     " configuration to complete...",
51                     max_parallel_configurations)
52             sleep(10)
53             continue
54
55         configuration = configurations.pop(0)
56         index = len(self.vars["run_ids"]) + 1
57
58         log.info("Saving configuration n. %d", index)
59         marshal.save(configuration, "configuration")
60
61         automl_config_artifact_id = kale_config_artifacts[index - 1].id
62         run_id = self._run_pipeline(index, {
63             "kale_dataset_id": str(kale_dataset_id),
64             "kale_config_id": str(automl_config_artifact_id)})
65         self.vars["run_ids"].append(run_id)
66         self._patch_context(run_id, index)

```

```

67
68     return self.vars["run_ids"]

```

As we've explained in sub-section 8.5.2 the `do_run` method contains the main code that runs in a Kale step. As a result, the inputs of the method correspond to the inputs of the Kale step, and the outputs of the method correspond to the outputs of the step. Therefore, let us give a more detailed explanation of the signature of the method:

- **Input Arguments:**

- **configurations:** A list of suggested pipeline configurations. This step will spin up a new Configuration Run for each one of these configurations. This list is actually an output from the previous step, which is passed using Kale's marshalling mechanism.
- **kale\_dataset\_id:** The ID of the Dataset Artifact (10.4.1.3) which corresponds to the input dataset of the problem. This is also an output from the previous step and it will be passed to each created Configuration Run. This way, since the Dataset Artifact represents (in MLMD) the actual input dataset, each Configuration Run will mark it as its input, sustaining a MLMD lineage for the experiment.
- **max\_parallel\_configurations:** An integer that expresses the maximum number of Configuration Runs that will can run in parallel. This is actually on of the input parameters in the `run_automl` API function.

- **Return Value:**

- **the list of Configuration Run IDs:** The next step will monitor the status of the Configuration Runs, so it will need their IDs to query the KFP server.

The main functionality of this step is implemented inside the reconciliation *while*-loop. This loop ensures that at each particular moment in time, there will be no more than `max_parallel_configurations` running. Here is a numbered list of steps that describes the logic inside the reconciliation loop:

1. **Compare the number of running Configuration Runs to the maximum allowed number of parallel Configuration Runs.** If the number of running Configuration Runs exceeds the maximum allowed number of parallel Configuration Runs, then sleep for a portion of time before you reenter the reconciliation loop. Otherwise, move on to the main body of the reconciliation loop.
2. **Extract the next configuration from the list of remaining configuration.**
3. **Save the configuration** in a marshal object in the volume of the Notebook Server. Later on in the process, Kale will take a snapshot of this volume which contains the configuration object. This cloned volume will be mounted in each pod of the corresponding Configuration Run. Subsequently, the Run will read and implement the pipeline described in the configuration.
4. **Spin up a new Configuration Run.** This is implemented by the `_run_pipeline` method of the `RunMetaLearningConfigurations` class. In a following section, we will expose the logic of this class.



5. **Add the Configuration Run ID in the list that holds all Configuration Run IDs.**  
This list is the only output of the *RunMetaLearningConfigurations* step. It will be used by the next step (*MonitorKFPRuns*) to query the KFP server and monitor the status of each individual Configuration Run.
6. **Update the MLMD Context of the Orchestrator** with links to the Contexts of the Configuration Runs. This way, we sustain a MLMD lineage between the AutoML Orchestrator and the Configuration Runs.

### 10.2.3.1 The `_run_pipeline` Method

As we described above, when the number of running Configuration Runs is less than the value of `max_parallel_configurations` (input parameter in the `run_automl` API function), the *RunMetaLearningConfigurations* step picks a configuration and spins up a new Configuration Run. The creation of the Configuration Run is implemented by the `_run_pipeline` method of the *RunMetaLearningConfigurations* class. The following list describes the signature of the method:

- **Input Arguments:**

- **index:** An integer that ranges from 1 to **number\_of\_configurations**. This integer is unique for the experiment, and it corresponds to the position of the configuration in the list of extracted configurations.
- **params:** A dictionary that contains the following fields:
  - \* **kale\_dataset\_id:** The ID of the Kale Dataset Artifact (10.4.1.3).
  - \* **kale\_config\_id:** The ID of the AutoMLConfiguration Artifact (subsection 10.4.1.2) which corresponds to the Configuration Run and which describes the corresponding pipeline configuration. To sustain a MLMD lineage for the AutoML experiment, we pass the *AutoMLConfiguration* Artifact ID to each individual Configuration Run. This way, the Run will be able to link this Artifact as its input by creating a MLMD Attribution (8.6) that links the MLMD Context of the Configuration Run with the Artifact. Linking an *AutoMLConfiguration* Artifact as input expresses the fact that the pipeline receives the corresponding configuration as its input from the AutoML Orchestrator.

These fields are to be added to the pipeline parameters of the newly created Configuration Run.

- **Return Value:**

- **the run ID** of the Configuration Run.

Listing 10.11: The `_run_pipeline` method of the *RunMetaLearningConfigurations* step class

```

1 def _run_pipeline(self, index, params: Dict = {}):
2     from kale.types import Param
3     from kale import PipelineConfig, Compiler
4     from kale.processors import PythonProcessor
5     from kale.ml.pipelines import sklearn_train_predict
6

```

```

7     volumes = rokutils.interactive_snapshot_and_get_volumes()
8     pipeline_config = PipelineConfig(
9         pipeline_name="sklearn-configuration-%d" % index,
10        experiment_name=kfputils.get_experiment_from_run_id(
11            kfputils.detect_run_uuid()).name,
12        marshal_path=self.marshal_path,
13        volumes=volumes)
14    pipeline_params = {k: Param(type(v).__name__, v)
15                       for k, v in params.items()}
16
17    log.newline()
18    log.info("Creating pipeline for configuration n. %d", index)
19    processor = PythonProcessor(
20        sklearn_train_predict.sklearn_train_predict, pipeline_config)
21    processor.pipeline.default_pipeline_parameters.update(pipeline_params)
22    pipeline = processor.run()
23    log.info("Running pipeline for configuration n. %d", index)
24    run = Compiler(pipeline).compile_and_run()
25    log.info("Successfully started run %s" % run.id)
26    log.newline()
27    return run.id

```

As we described above, `_run_pipeline` essentially creates actual KubeFlow Pipelines, called Configuration Runs. For this purpose this, this method mainly uses Kale's backend (section 8.5). Here is a numbered list of steps that describes the main logic inside the `_run_pipeline` method:

1. **It takes a snapshot of the volume that is mounted in the Pod.** This volume, will be mounted to the first step of each Configuration Run, and it contains a marshal object with the corresponding pipeline configuration.
2. **It instantiates a `kale.PipelineConfig` object** with all the basic descriptive information (such as the name) of the Configuration Run.
3. **It creates a dictionary with `kale_dataset_id` and `kale_config_id` `Param` objects** that will be passed as pipeline parameters to the Configuration Run.
4. **It instantiates a `kale.processors.PythonProcessor` (8.5.5)** with a Python function written in Kale DSL code (8.5.4). This Python function essentially describes the entire ML functionality of the Configuration Run, and we will expose it in a following section. The `PythonProcessor` evaluates this pipeline function and returns a Pipeline object (8.5.3).
5. **It compiles the Pipeline object into an Argo Workflow and applies it to Kubernetes.** This happens using the `kale.Compiler` object and its `compile_and_run` method (8.5.6).

#### 10.2.4 The `MonitorKFPRuns` Step

In this subsection, we will expose and explain the mechanism that the `MonitorKFPRuns` step implements. In summary, this step watches the statuses of the Configuration

Runs that were created by the previous step (*run-metalearning-configurations*), and when they are all finished, either successfully or unsuccessfully, this step completes its execution. Let us view the code that runs inside this Kale step, and more specifically its class definition and *do\_run* method:

Listing 10.12: The class definition and *do\_run()* method of the *monitor-kfp-runs* step

```

1  class MonitorKFPRuns(Step):
2      """Wait for KFP pipelines to complete.
3
4      Ins:
5          run_ids
6
7      Outs:
8          run_ids
9      """
10     name = "monitor-kfp-runs"
11     outs = Param.odict(["run_ids"], step_name=name)
12
13     def do_run(self, run_ids):
14         """Implementation of MonitorKFPRuns."""
15         from time import sleep
16
17         log.info("Monitoring runs: %s", run_ids)
18         statuses = {run_id: "Pending" for run_id in run_ids}
19
20         # Add custom properties linking the MLMD Execution with the runs to
21         # monitor
22         log.info("Patching MLMD Execution custom properties with the"
23                 " configuration run IDs...")
24         mlmd = mlmdutils.get_mlmd_instance()
25         custom_props = {"configuration_run_%d" % idx:
26                         kfputils.format_kfp_run_id_uri(run_ids[idx])
27                         for idx in range(len(run_ids))}
28         mlmdutils.patch_execution_custom_properties(mlmd.execution.id,
29                                                    custom_props)
30         log.info("Successfully patched MLMD Execution")
31
32         while any(map(lambda state: state not in kfputils.KFP_RUN_FINAL_STATES,
33                       statuses.values())):
34             log.newline()
35             log.info("Updating pipelines statuses...")
36             for run_id in statuses.keys():
37                 if statuses[run_id] not in kfputils.KFP_RUN_FINAL_STATES:
38                     statuses[run_id] = kfputils.get_run(run_id).run.status
39                     log.info("Run '%s': %s", run_id, statuses[run_id])
40             sleep(5)
41
42         log.info("All done!")

```

```
43     return run_ids
```

As we can see from the code above, the signature of the `do_run` method of the step is fairly simple. It receives the list of Configuration Run IDs as input, and it returns it for the next step to consume. The input list of IDs will be used to query the KFP server for the statuses of the Configuration Run pipelines. Let us view **a numbered list of steps** that describes the mechanism behind `MonitorKFPRuns`'s `do_run` method:

1. **Update the custom properties of the MLMD Execution with the Configuration Run IDs to monitor.** As we mentioned in (8.6), executions are mapped to pipeline steps. Therefore, since the `MonitorKFPRuns` step monitors all the Configuration Runs of the experiment, we link this step to these Configuration Runs, in order to sustain an MLMD lineage across our AutoML experiment. For this purpose, we use Kale's `mlmdutils` utility module.
2. **Keep a dictionary of the statuses of all Configuration Runs.** Initially, the status of each Configuration Run is set to `Pending`. Configuration Runs are represented by their run IDs.
3. **Sleep while not all Configuration Runs are in a final state.** We consider a KubeFlow Pipeline to be in a final state when its status is: `Succeeded`, `Skipped`, `Failed` or `Error`.
4. **When all Configuration Runs are finished, return their run IDs.** This list of run IDs is essentially the only return parameter of the step. The next step of the AutoML Orchestrator (`GetBestConfiguration`) will need these run IDs to query the KFP server for the Configuration Runs and get the metrics they produced.

### 10.2.5 The `GetBestConfiguration` Step

In this subsection, we will expose and explain the mechanism that the `get-best-configurations` step implements. In summary, this step takes as input the list of Configuration Run IDs, collects the metric scores of the Configuration Runs that succeeded, finds the Configuration Run with the best score and returns the corresponding configuration object. Let us view the class definition and the `do_run` method of the step:

Listing 10.13: The class definition and `do_run()` method of the `get-best-configuration` step

```
1 class GetBestConfiguration(Step):
2     """Get the best-performing MetaLearning configuration.
3
4     Ins:
5         run_ids
6         configurations
7         metric
8
9     Outs:
10        best_configuration
11        """
12     name = "get-best-configuration"
```

```

13     outs = Param.odict({"best_configuration"}, step_name=name)
14
15     def do_run(self, run_ids, configurations, metric):
16         """Implementation of GetBestConfiguration."""
17         metrics = dict()
18         for run_id in run_ids:
19             log.info("Collecting metrics for run: %s", run_id)
20             metrics[run_id] = kfputils.get_kfp_run_metrics(run_id)
21         final_metrics = metrics.copy()
22
23         log.newline()
24         log.info("Collected metrics: \n")
25         for run_id, _metrics in metrics.items():
26             log.info("  Run %s:", run_id)
27             if not _metrics.values():
28                 log.info("    No metrics found.")
29                 del final_metrics[run_id]
30                 for name, value in _metrics.items():
31                     log.info("      %s: %s", name, value)
32             log.info("Using metric '%s' as target metric.", metric.name)
33
34         # Get best metric, excluding empty metrics dictionaries
35         opt = max if metric._sign == 1 else min # see arrikto/dev#1128
36         best_run_uid, best_metric = opt(final_metrics.items(),
37                                       key=lambda x: x[1][metric.name])
38         log.info("Best run id: %s", best_run_uid)
39
40         log.info("Patching MLMD Execution and Context custom properties with"
41                " the best configuration run ID...")
42         custom_prop = {"best_configuration_run":
43                       kfputils.format_kfp_run_id_uri(best_run_uid)}
44         mlmd = mlmdutils.get_mlmd_instance()
45         mlmdutils.patch_execution_custom_properties(mlmd.execution.id,
46                                                    custom_prop)
47         log.info("Successfully patched MLMD Execution")
48         mlmdutils.patch_context_custom_properties(mlmd.run_context.id,
49                                                  custom_prop)
50         log.info("Successfully patched MLMD Context")
51
52         best_configuration = configurations[run_ids.index(best_run_uid)]
53         return best_configuration

```

The process of finding the best scoring configuration is essentially implemented inside the **do\_run** method. The following list describes the signature of the method:

- **Input Arguments:**

- **run\_ids:** A list of the Configuration Run IDs of the experiment. This is actually the output of the previous, *MonitorKFPRuns* step. Each of these IDs will be used to collect the metric score of the corresponding Configuration Run.

- **configurations:** A list of the suggested pipeline configurations. This is actually the output of the *GetMetaLearningConfigurations* step. The *GetBestConfiguration* step extracts and returns the best pipeline configuration out of that list.
  - **metric:** The metric that is used to evaluate the results of each Configuration Run. This is actually the function object that the user provided as input to the *run\_automl* API function (section 10.1).
- **Return Value:**
    - **configuration:** The highest scoring configuration in the AutoML process.

Here is a **numbered list of steps** that describes the mechanism behind *GetBestConfiguration*'s **do\_run** method:

1. **Collect the metric scores of the Configuration Runs:** This happens by iterating through the list of run IDs and collecting each metric score using the *get\_kfp\_run\_metrics* function from Kale's *kfputils* module.
2. **Find the Configuration Run (and configuration) that scored the highest** by iterating through the run IDs.
3. **Update the custom properties of the MLMD Execution and Context with the ID of the best scoring Configuration Run.** As we mentioned in section 8.6, executions are mapped to pipeline steps. Therefore, since the *GetBestConfiguration* step finds the highest scoring Configuration Run, we link this step to that Configuration Run, so that we sustain a MLMD lineage across our AutoML experiment. For this purpose, we use Kale's *mlmdutils* utility module.
4. **Return the configuration that corresponds to the best scoring Configuration Run.**

### 10.2.6 The *RunKatibExperiment* Step

In this subsection, we will expose the mechanism that the *run-katib-experiment* step implements. In summary, *run-katib-experiment* takes the highest scoring configuration from the previous step, and creates a Katib experiment (8.4.4) to further optimize this top-scoring configuration by finding hyper-parameter values that produce even better results than the starting configuration.

This and the following step (*monitor-katib-experiment*) are executed **only in the case that the user has specified a tuner** (*katib.ExperimentSpec* object) input parameter in the *run\_automl* function call (section 10.1). If no *tuner* is passed in the *run\_automl* API function, then the *get-best-configuration* step (subsection 10.2.5) is the last step of the AutoML Orchestrator pipeline. One can also conclude this conditional execution of the *run-katib-experiment* step by looking at listing 10.2, where these two steps are executed only if pipeline parameter *hp\_tune* is set to "true".

Let us view the class definition and the main methods of the step:

Listing 10.14: The class definition and methods of the *run-katib-experiment* step

```

1 class RunKatibExperiment(Step):
2     """Run a Katib experiment.
3
4     Ins:
5         tuner (katib.V1beta1ExperimentSpec): Experiment spec
6         best_configuration (ConfigSpace.Configuration):
7         metric (autosklearn.metric): An (Auto)SKLearn metric
8         kale_dataset_id (int): The MLMD artifact ID
9
10    Outs:
11        katib_experiment_name (str): Katib experiment name
12    """
13    name = "run-katib-experiment"
14    outs = Param.odict(["katib_experiment_name"], step_name=name)
15    actions = ["KatibExperiment"]
16
17    def _get_hyperparams(self, configuration):
18        return {
19            hp_name: configuration[hp_name]
20            for hp_name in configuration.keys()
21            if (any(map(lambda alg_type: hp_name.startswith(alg_type),
22                       ["classifier", "regressor"]))
23                and not hp_name.endswith("__choice__"))}
24
25    def _generate_hyperparam_conf(self, key, value, conf_space):
26        conf_space = copy.deepcopy(conf_space)
27        hp = conf_space.get_hyperparameter(key)
28        if hasattr(hp, "upper") or hasattr(hp, "choices"):
29            hp.default_value = value
30        return hp
31    return None
32
33    def _get_hyperparams_confs(self, configuration):
34        configuration = copy.deepcopy(configuration)
35        hyperparams_conf = []
36        for key, value in self._get_hyperparams(configuration).items():
37            hp = self._generate_hyperparam_conf(
38                key, value, configuration.configuration_space)
39            if hp:
40                hyperparams_conf.append(hp)
41        return hyperparams_conf
42
43    def _get_param(self, conf):
44        from kubeflow import katib
45        if hasattr(conf, "choices"):
46            return katib.V1beta1ParameterSpec(
47                feasible_space=katib.V1beta1FeasibleSpace(

```

```

48         list=list(conf.choices)),
49         name=self._conf_name(conf),
50         parameter_type="categorical")
51     else: # for now assume just float ranges
52         return katib.V1beta1ParameterSpec(
53             feasible_space=katib.V1beta1FeasibleSpace(
54                 max=str(float(conf.upper)),
55                 min=str(float(conf.lower)),
56                 # heuristic just for test purposes
57                 step=str((float(conf.upper) - float(conf.lower)) / 10)),
58             name=self._conf_name(conf),
59             parameter_type="double")
60
61     def do_run(self, tuner, best_configuration, metric, kale_dataset_id):
62         """Implementation of RunKatibExperiment."""
63         from kubeflow import katib
64         from kale.types import Param
65         from kale.processors import PythonProcessor
66         from kale import marshal, PipelineConfig, Compiler
67         from kale.ml.pipelines import sklearn_train_predict
68
69         hyperparam_confs = self._get_hyperparams_confs(best_configuration)
70
71         # Marshal the configuration, used by the train-predict pipeline
72         marshal.save(best_configuration, "configuration")
73
74         # Configure the HP tuning settings
75         tuner.algorithm = katib.V1beta1AlgorithmSpec(algorithm_name="grid")
76         tuner.objective = katib.V1beta1ObjectiveSpec(
77             objective_metric_name=metric.name,
78             type=tuner.objective.type or "maximize")
79         tuner.parameters = [self._get_param(conf) for conf in hyperparam_confs]
80
81         volumes = rokutils.interactive_snapshot_and_get_volumes()
82         log.info("Creating parametrized pipeline...")
83         pipeline_config = PipelineConfig(
84             pipeline_name="katib-trial-sklearn-configuration",
85             experiment_name=kfputils.get_experiment_from_run_id(
86                 kfputils.detect_run_uuid()).name,
87             marshal_path=self.marshal_path,
88             volumes=volumes,
89             katib_metadata=tuner,
90             katib_run=True)
91         processor = PythonProcessor(
92             sklearn_train_predict.sklearn_train_predict, pipeline_config)
93         pipeline = processor.run()
94
95         for conf in hyperparam_confs:

```



```

96     pipeline.default_pipeline_parameters[
97         self._conf_name(conf)] = Param(name=self._conf_name(conf),
98                                         param_type="str")
99
100     def _patch_step_cli(step: BaseStep):
101         del step.ins["masked_inputs"]
102         for conf in hyperparam_confs:
103             name = ".msk.%s" % self._conf_name(conf)
104             step.ins[name] = Param(name=name)
105     _patch_step_cli(pipeline.get_step("run-sklearn-transformer"))
106     _patch_step_cli(pipeline.get_step("train-sklearn-estimator"))
107
108     log.info("Running Katib experiment...")
109     experiment = Compiler(pipeline).compile_and_run()
110     log.info(experiment)
111
112     self._patch_experiment_uri(experiment["metadata"]["name"],
113                               experiment["metadata"]["namespace"])
114     return experiment["metadata"]["name"]

```

The `do_run` method of the `RunKatibExperiment` step class, which contains the ML-oriented code that gets executed in the step, works as follows:

1. **It gets the top-scoring configuration as input** from the previous step.
2. **Extracts the names of its hyper-parameters and gets their feasible space.** This happens using the `_get_hyperparams_confs` method.
3. **Compiles and runs a Katib experiment using the pipeline specified in the `sklearn_train_predict` pipeline function.** We analyzed the structure of this pipeline function in section 10.3.1. Katib, later in the process, will run several such pipelines with different hyper-parameter values to figure out how the hyper-parameters affect their scores.
4. **Returns the name of the created Katib experiment, so that the next step can monitor its execution.**

### 10.2.7 The `MonitorKatibExperiment` Step

This is the last step of the AutoML Orchestrator pipeline and its purpose is to monitor the Katib experiment that the previous step created.

Listing 10.15: The class definition and `do_run` method of the `run-katib-experiment` step

```

1  class MonitorKatibExperiment(Step):
2      """Wait for a Katib experiment to complete.
3
4      Ins:
5          katib_experiment_name (str): Name of the Katib experiment to monitor.
6      """
7      name = "monitor-katib-experiment"
8

```

```

9     def do_run(self, katib_experiment_name: str):
10         """Implementation of MonitorKatibExperiment."""
11         from kale.common import katibutils
12
13         katibutils.wait_for_hptuning_experiment(katib_experiment_name)

```

As we can see from above, the `do_run` method of the `MonitorKatibExperiment` step class simply waits for the Katib experiment to finish its execution. It does that by calling the `wait_for_hptuning_experiment` utility function from Kale's `common.katibutils` module. Here is the code of this function:

Listing 10.16: The `wait_for_hptuning_experiment` utility function from the `kale.common.katibutils` module

```

1  def wait_for_hptuning_experiment(experiment_name: str):
2      """Wait for an HP Tuning experiment to succeed.
3
4      Args:
5          experiment_name (str): Name of the HP Tuning experiment.
6
7      Returns:
8          tuple(str, str, str): Status, Condition reason, Condition message.
9      """
10     def sleep_with_progress(total, interval, msg):
11         for i in range(0, total // interval):
12             log.info("%s %d...", msg, total - i * interval)
13             time.sleep(interval)
14
15     while True:
16         log.newline(2)
17         log.info("Watching for HP Tuning experiment: '%s'",
18                 experiment_name)
19         sleep_with_progress(30, 5, "Checking status in")
20
21         experiment = get_experiment(experiment_name, podutils.get_namespace())
22         status = get_experiment_status(experiment["status"])
23         log.info("Experiment status: %s", status)
24         if status[0] not in EXPERIMENT_FINAL_STATES:
25             continue
26         return status

```

### 10.3 The Configuration Run

In this section, we will describe the architecture of Configuration Runs. These are the pipelines that the AutoML Orchestrator creates in its `RunMetaLearningConfigurations` step (10.2.3) to train models in parallel. Each Configuration Run in an AutoML experiment implements one of the pipeline configurations that auto-sklearn extracts.

These pipelines essentially consist of three steps, and each step corresponds to a

specific part of a ML workflow. Here's an overview of each step:

1. **run-sklearn-transformer**: This step implements the preprocessor specified in a pipeline configuration. Its purpose is to process the input dataset (both train and test set) and bring it to a state that will allow the model to efficiently be trained and tested on it.
2. **train-sklearn-estimator**: This step essentially implements the actual model architecture that a configuration describes. Its purpose is to produce a trained model using the processed train set from the previous step.
3. *infer-sklearn-predictor*: This step uses the processed test set from step *run-sklearn-transformer* to test the trained model from step **train-sklearn-estimator** using the metric function that the user provided in *run\_automl()*. When the metric scores from this step are finally produced, the entire Configuration Run finishes its execution.

To prove that the architecture we provided above holds, let us show the following image that showcases an example of a completed Configuration Run, as shown in the KubeFlow Pipelines UI.

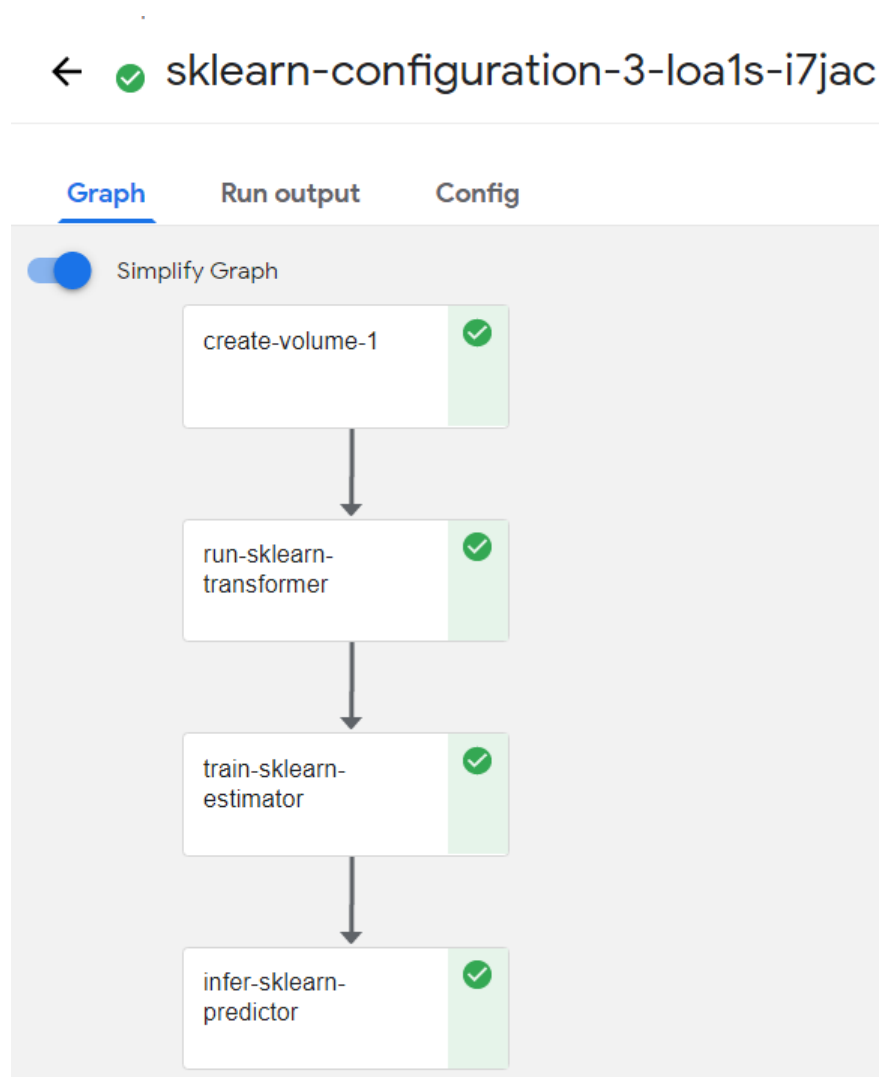


Figure 10.2: Example of completed Configuration Run in the KFP UI

As we mentioned in sub-section 10.2.3, the pipeline function that is used as a template for the Configuration Runs is the `sklearn_train_predict` function from the `kale.ml.pipelines` module. Let us expose and analyze this pipeline function in the following sub-section.

### 10.3.1 The `sklearn_train_predict` Pipeline Function

The `sklearn_train_predict` function is a Python pipeline function, written in Kale's domain-specific language (8.5.4), that essentially describes the structure of the Configuration Run that is created by the `RunMetaLearningConfigurations` (10.2.3) step of the AutoML Orchestrator. Since it's written in Kale DSL, it uses `Kale Step` (8.5.2) instances to represent steps in the pipeline. Here is the corresponding code snippet:

Listing 10.17: The `sklearn_train_predict` pipeline function

```

1  def sklearn_train_predict(kale_dataset_id="-1", kale_config_id="-1"):
2      """Train and validate a SKLearn model from an AutoML configuration."""
3      (x_processed,
4       x_test_processed,
5       kale_dataset_id_local,
6       transformer_pipeline) = RunSKLearnTransformer()(
7          ml_assets_marshall_path("configuration.dillpkl"),
8          ml_assets_marshall_path("dataset.dillpkl"),
9          kale_dataset_id,
10         kale_config_id,
11         "{}"
12     )
13     model, kale_model_id = TrainSKLearnEstimator()(
14         ml_assets_marshall_path("configuration.dillpkl"),
15         x_processed,
16         ml_assets_marshall_path("dataset.dillpkl"),
17         kale_dataset_id_local,
18         "{}"
19     )
20     InferSKLearnPredictor()(model,
21                             x_processed,
22                             x_test_processed,
23                             ml_assets_marshall_path("metric.joblib"),
24                             ml_assets_marshall_path("dataset.dillpkl"),
25                             ml_assets_marshall_path("task.dillpkl"),
26                             kale_model_id,
27                             kale_dataset_id_local)

```

As we see from the previous code snippet, the Configuration Run pipeline that the `sklearn_train_predict` describes, consists of three Step classes:

1. **RunSKLearnTransformer**
2. **TrainSKLearnEstimator**
3. **InferSKLearnPredictor**

Each of these is a regular Kale step that implements a specific part of the Configuration Run pipeline. Followingly, we will expose the functionality of each one of these steps.

### 10.3.2 The *RunSKLearnTransformer* Step

In this subsection, we will expose and explain the mechanism that the *run-sklearn-transformer* step implements. We will describe how this step reads an input configuration and implements the preprocessor step that this configuration describes. Let us view the class definition and the main methods of the step:

Listing 10.18: The class definition and the main methods of the *run-sklearn-transformer* step

```

1  class RunSKLearnTransformer(PatchMaskedInputsMixin, Step):
2      """Run a SKLearn transformer over a dataset.
3
4      Ins:
5          configuration
6          dataset
7          kale_dataset_id
8          kale_config_id
9
10     Outs:
11         x_processed
12         x_test_processed
13         kale_dataset_id_local
14         transformer_pipeline
15
16     MLMD Inputs:
17         kale.Configuration
18         kale.Dataset
19
20     MLMD Outputs:
21         kale.Transformer
22     """
23     name = "run-sklearn-transformer"
24     outs = Param.odict({"x_processed", "x_test_processed",
25                        "kale_dataset_id_local", "transformer_pipeline"},
26                       step_name=name)
27
28     def _submit_dataset_artifact(self, dataset):
29         from kale.settings import settings
30         from kale.marshall.utils import strip_marshal_path
31
32         dataset_artifact = dataset.as_artifact()
33         rok_version = self.vars.get("autosnapshot_start")
34         if not dataset.artifact_uri and rok_version:
35             dataset.artifact_uri = rokutils.get_uri_in_version(
36                 rok_version, strip_marshal_path(settings.INS["dataset"]))

```

```

37     dataset_artifact = dataset.submit_artifact()
38     return dataset_artifact
39
40     def _link_input_artifacts(self, kale_dataset_id, kale_config_id):
41         mlmd = mlmdutils.get_mlmd_instance()
42         if kale_config_id > 0:
43             mlmd.link_artifact_as_input(kale_config_id)
44         if kale_dataset_id > 0:
45             mlmd.link_artifact_as_input(kale_dataset_id)
46
47     def _link_output_artifacts(self):
48         from kale.common.artifacts import Transformer
49
50         mlmd = mlmdutils.get_mlmd_instance()
51         self.vars["transformer_artifact"] = Transformer(
52             self.name, self.vars["preprocessor"]).submit_artifact()
53         mlmd.link_artifact_as_output(self.vars["transformer_artifact"].id)
54
55     def do_run(self, configuration, dataset, kale_dataset_id, kale_config_id,
56               masked_inputs):
57         """Implementation of RunSKLearnTransformer."""
58         import json
59         import sklearn
60         from kale.ml import utils
61
62         kale_dataset_id = int(kale_dataset_id)
63         kale_config_id = int(kale_config_id)
64         if kale_dataset_id <= 0:
65             kale_dataset_id = self._submit_dataset_artifact(dataset).id
66         self._link_input_artifacts(kale_dataset_id, kale_config_id)
67
68         configuration = utils.patch_configuration(configuration,
69                                                  json.loads(masked_inputs),
70                                                  coerce=True)
71         steps = utils.get_sklearn_steps_list(configuration)
72         self.vars["preprocessor"] = sklearn.pipeline.Pipeline(steps[:-1])
73         x_processed = self.vars["preprocessor"].fit_transform(dataset.features,
74                                                            dataset.targets)
75         x_test_processed = self.vars["preprocessor"].transform(
76             dataset.features_test)
77
78         self._link_output_artifacts()
79         return (x_processed, x_test_processed, kale_dataset_id,
80               self.vars["preprocessor"])

```

As in the case of the AutoML Orchestrator steps, the *do\_run()* method is the main code that runs inside the step. Consequently, let us provide an overview on the logic that this method implements. The *do\_run* method:

- 1. Links the Dataset and AutoMLConfiguration Artifacts as inputs of this step.** This step takes as inputs the IDs of the Kale Dataset and AutoMLConfiguration Artifacts (pipeline parameters), so the linking is implemented via the `_link_input_artifacts` method of the step.
- 2. Reads the input configuration.** If the Configuration Run is part of a Katib experiment (created by the `run-katib-experiment` step) then we update the input configuration, created by auto-sklearn, with the values that are suggested by Katib. In the case that the Configuration Run is created by the `run-metalearning-configurations` step, we keep the configuration input as is. We implement this decision in the `patch_configuration` function of the `kale.ml.utils` module:

Listing 10.19: The `patch_configuration` function of the `kale.ml.utils` module

```

1  def patch_configuration(configuration: Configuration,
2                          overrides: Dict[str, Any],
3                          coerce=False):
4      """Patch a configuration.
5
6      Args:
7          configuration: A suggested configuration extracted by auto-sklearn.
8          overrides (dict): Override configuration values by providing them with
9                          keys matching the last token in the original Configuration keys.
10                     Provide an empty dictionary to make this function a no-op.
11          coerce (bool): Convert the override value to the destination type.
12
13      Returns:
14          Configuration: Patched configuration.
15      """
16      log.info("Patching base configuration...")
17      log.info("Base %s", configuration)
18
19      if not overrides:
20          log.info("No input parameters found to patch the base"
21                  " configuration.")
22          return configuration
23
24      log.info("Using the following configs to patch the base"
25              " configuration: %s", overrides)
26      patched_config = copy.deepcopy(configuration)
27      for name, override_value in overrides.items():
28          for key, conf_value in patched_config.get_dictionary().items():
29              if key.split(":-1"] == name:
30                  if coerce:
31                      override_value = type(conf_value)(override_value)
32                  patched_config[key] = override_value
33      log.info("Patched %s", patched_config)
34      return patched_config
35

```

3. **Transforms the configuration into an ordered list of steps.** For this purpose, we implemented the `get_sklearn_steps_list` function of the `kale.ml.utils` module:

Listing 10.20: The `get_sklearn_steps_list` function of the `kale.ml.utils` module

```

1  def get_sklearn_steps_list(config: Configuration):
2  """Return a list of steps for sklearn Pipeline."""
3  log.info("Getting SKLearn steps list...")
4  if config.get("classifier:__choice__"):
5      p = SimpleClassificationPipeline(config)
6  elif config.get("regressor:__choice__"):
7      p = SimpleRegressionPipeline(config)
8  else:
9      raise ValueError("No model found in input configuration")
10 return p.steps

```

Essentially, this function initializes either a `SimpleClassificationPipeline` or `SimpleRegressionPipeline` from the `autosklearn.pipeline` (9.2.5) module and returns its steps list-attribute.

4. **Removes the last step that corresponds to the estimator.** Since the list of steps is ordered, the last step corresponds to the estimator, and all the previous steps are preprocessing steps.
5. **Creates an `sklearn.pipeline.Pipeline` object with the list of preprocessing steps.**
6. **Processes the dataset using the `transform` and `fit_transform` methods** (8.1.1) of the sklearn Pipeline object.
7. **Creates a Kale Transformer Artifact** (10.4.1.3) and links it as output of the step using the `_link_output_artifacts` method.
8. **Returns the processed dataset for the next step to consume.**

### 10.3.3 The `TrainSKLearnEstimator` Step

In this subsection, we will expose and explain the mechanism that the `train-sklearn-estimator` step implements. In general, the `do_run` method of this step follows the same logic as the `do_run` of the previous step. We will describe how `train-sklearn-estimator` reads the input configuration, creates the model that the configuration describes and then trains the model on the processed training set that the previous step created. Let us view the class definition and the main methods of the step:

Listing 10.21: The class definition and the main methods of the `train-sklearn-estimator` step

```

1  class TrainSKLearnEstimator(PatchMaskedInputsMixin, Step):
2  """Train a SKLearn estimator from an AutoML configuration.
3
4  Ins:

```



```

5     configuration:
6     x_processed:
7     dataset:
8     kale_dataset_id_local:
9
10    Outs:
11        model: Trained model
12        kale_model_id: MLMD artifact ID of the trained model
13
14    MLMD Inputs:
15        kale.Configuration
16
17    MLMD Outputs:
18        kale.Model
19    """
20    name = "train-sklearn-estimator"
21    outs = Param.odict(["model", "kale_model_id"], step_name=name)
22
23    def _link_input_artifact(self, kale_dataset_id):
24        mlmd = mlmdutils.get_mlmd_instance()
25        mlmd.link_artifact_as_input(kale_dataset_id)
26
27    def _link_output_artifacts(self, model):
28        from kale.common.artifacts import Model
29
30        mlmd = mlmdutils.get_mlmd_instance()
31        self.vars["kale_model_artifact"] = Model(
32            model, context_name=self.name).submit_artifact()
33        mlmd.link_artifact_as_output(self.vars["kale_model_artifact"].id)
34
35    def do_run(self, configuration, x_processed, dataset,
36              kale_dataset_id_local, masked_inputs):
37        """Implementation of TrainSKLearnEstimator."""
38        import json
39        from kale.ml import utils
40
41        self._link_input_artifact(kale_dataset_id_local)
42
43        configuration = utils.patch_configuration(configuration,
44                                                json.loads(masked_inputs),
45                                                coerce=True)
46        self.vars["model"] = utils.get_sklearn_steps_list(configuration)[-1][1]
47
48        # NOTE: The 'model' variable is important, because we use this very
49        # same marshalled model to also start KFServing servers, which expects
50        # to find 'model.joblib' files.
51        self.vars["model"].fit(x_processed, dataset.targets)
52

```

```

53     self._link_output_artifacts(self.vars["model"])
54     return self.vars["model"], self.vars["kale_model_artifact"].id

```

As in the case of the AutoML Orchestrator steps, the `do_run()` method is the main code that runs inside the step. Consequently, let us provide an overview on the logic that this method implements. The `do_run` method:

1. **Links the Kale Dataset Artifact as input of this step.** The step takes as inputs the ID of the Kale Dataset from the previous step (*run-sklearn-transformer*), and links it as its input using the `_link_input_artifacts` method of the class, in order to sustain a MLMD lineage for the Configuration Run.
2. **Reads the input marshalled configuration.** If the Configuration Run is part of a Katib experiment (created by the *run-katib-experiment* step) then the `do_run` method updates the initial input configuration, created by auto-sklearn, with the values that are suggested by Katib. Otherwise, if the Configuration Run is created by the *run-metalearning-configurations* step, we keep the input configuration as is. This decision is implemented in the `patch_configuration` function of the `kale.ml.utils` module. We exposed this function in listing 10.19.
3. **Transforms the configuration into an ordered list of steps.** For this purpose, we implemented the `get_sklearn_steps_list` function of the `kale.ml.utils` module. We exposed this function in listing 10.20. Essentially, `get_sklearn_steps_list` initializes either a *SimpleClassificationPipeline* or *SimpleRegressionPipeline* from the *autosklearn.pipeline* (9.2.5) module and returns its `steps` list-attribute.
4. **Keeps the last step of the pipeline configuration which corresponds to the estimator.** Since the list of steps is ordered, the last step corresponds to the estimator, and all the previous steps are preprocessing steps. Later on, we will create a model object out of this estimator step, which we will train using the training set.
5. **Creates an *sklearn.pipeline Pipeline* object using the estimator step of the pipeline configuration.**
6. **Trains the Pipeline object on the processed training dataset using its *fit* methods** (8.1.1). The returned object is our trained model.
7. **Creates a Kale Model Artifact (10.4.1.3)** and links it as output of the step using the `_link_output_artifacts` method.
8. **Returns the model object for the next step to consume.** The next step (*infer-sklearn-predictor*) and it will test the returned model using the processed test dataset that *run-sklearn-transformer* created.

### 10.3.4 The *InferSKLearnPredictor* Step

In this sub-section, we will expose and explain the mechanism that the *InferSKLearnPredictor* step implements. In general, the `do_run` method of this step reads the input trained model that was created by the previous step, and then tests the model on the

processed test set that the *RunSKLearnTransformer* step created. Let us view the class definition and the main methods of the step:

Listing 10.22: The class definition and the main methods of the *infer-sklearn-predictor* step

```

1  class InferSKLearnPredictor(Step):
2      """Run predictions on a trained model.
3
4      Ins:
5          model
6          x_test_processed
7          metric
8          dataset
9          task
10         kale_model_id
11         kale_dataset_id_local
12
13     MLMD Inputs:
14         kale.Model
15         kale.Dataset
16
17     MLMD Outputs:
18         kale.TensorboardLogs
19     """
20     name = "infer-sklearn-predictor"
21     has_metrics = True
22
23     def _link_input_artifacts(self, kale_dataset_id, kale_model_id):
24         mlmd = mlmdutils.get_mlmd_instance()
25         mlmd.link_artifact_as_input(kale_dataset_id)
26         mlmd.link_artifact_as_input(kale_model_id)
27
28     def _update_model_artifact(self, kale_model_id, metric_name, metric_value):
29         mlmd = mlmdutils.get_mlmd_instance()
30         model = mlmdutils.get_artifact_by_id(kale_model_id)
31         model_metrics = json.loads(model.properties["metrics"].string_value)
32         model_metrics.update({metric_name: metric_value})
33
34         model_name = model.properties["name"].string_value
35         class_name = model_name.split("/")[-1]
36         new_name = "%s/%s" % (mlmd.name or "", class_name)
37
38         mlmdutils.patch_artifact_properties(kale_model_id,
39                                             {"name": new_name,
40                                              "metrics": model_metrics})
41
42     def _produce_report(self, task, dataset, model):
43         from kale.ml import visualizations

```

```

44     from tensorboardX import SummaryWriter
45
46     log.info('Logging report to Tensorboard...')
47
48     viz = visualizations.SklearnVisualizer(task=task, dataset=dataset,
49                                           model=model.choice.estimator,
50                                           is_fitted=True)
51     report = viz.plot_report()
52     writer = SummaryWriter(log_dir="logs/experiment")
53     for name, figure in report.items():
54         writer.add_figure(name, figure)
55     log.info("Successfully logged report to Tensorboard")
56     self.vars["tb_logs_dir"] = os.path.realpath("logs")
57
58     def _log_tb_logs_artifact(self):
59         from kale.common.artifacts import TensorboardLogs
60
61         rok_version = self.vars.get("autosnapshot_end")
62         uri = self.vars["tb_logs_dir"]
63         if os.path.exists(uri) and rok_version:
64             uri = rokutils.get_uri_in_version(rok_version,
65                                             self.vars["tb_logs_dir"])
66         tb_logs = TensorboardLogs(name=self.name)
67         tb_logs.artifact_uri = uri
68         tb_logs_artifact = tb_logs.submit_artifact()
69
70         mlmd = mlmdutils.get_mlmd_instance()
71         mlmd.link_artifact_as_output(tb_logs_artifact.id)
72
73     def do_run(self, model, x_processed, x_test_processed, metric, dataset,
74              task, kale_model_id, kale_dataset_id_local):
75         """Implementation of InferSKLearnPredictor."""
76         import autosklearn.metrics as metrics
77         from kale.sdk.logging import log_metric
78         from kale.ml.utils import extract_task_type
79
80         self._link_input_artifacts(kale_dataset_id_local, kale_model_id)
81
82         predictions = model.predict(x_test_processed)
83         task_type = extract_task_type(dataset.targets_test, task)
84
85         # 'calculate_score' returns negative values if "lower is better" for
86         # the given metric. For this reason, we will get the absolute number
87         # of the result.
88         metric_value = abs(metrics.calculate_score(dataset.targets_test,
89                                                  predictions, task_type,
90                                                  metric))
91         log.info("Metric '%s': %s", metric.name, metric_value)

```

```

92     log_metric(metric.name, metric_value)
93
94     self._update_model_artifact(kale_model_id, metric.name, metric_value)

```

As in the case of the AutoML Orchestrator steps, the *do\_run()* method is the main code that runs inside the step. Consequently, let us provide an overview on the logic that this method implements. The *do\_run* method:

1. **Links the Kale Dataset Artifact and the Kale Model Artifact as inputs of this step.** The step takes as inputs the ID of the Kale Dataset from the previous step (*run-sklearn-transformer*), and links it as its input using the *\_link\_input\_artifacts* method of the class, in order to sustain a MLMD lineage for the Configuration Run. The same thing happens for the Kale Model Artifact as well.
2. **Produces predictions using the input model object and the input test set.** The model object is an output of the *train-sklearn-estimator*, whereas the processed test set is an output of the *run-sklearn-transformer*. The model is essentially a sklearn *Pipeline* object (8.1.1), so we it produces its predictions on a test dataset via its *predict* method.
3. **Calculates the metric score of the predictions.** This happens using the *calculate\_score* function of auto-sklearn's *metrics* module.
4. **Updates the "metrics" property of the Kale Model Artifact with the metric score that was calculated.** This happens using the *\_update\_model\_artifact* method of the class.
5. **Returns the model object for the next step to consume.** The next step (*infer-sklearn-predictor*) and it will test the returned model using the processed test dataset that *run-sklearn-transformer* created.

## 10.4 The MLMD Lineage and Status Tracking of Kale-AutoML Experiments

During the development of our AutoML process with Kale, we wanted to create an end-to-end lineage of our AutoML experiments. The end goal was to enable the user, that creates a *run\_automl* function call, to view an entire summary of the created AutoML experiment. This entails:

- **the status of the AutoML Orchestrator**
- **the status of the Configuration Runs**
- **the status of the Katib Experiment, if a tuner input is provided**

We aimed to be able to track all of the above, from a single *run\_automl* API call provided with the KFP run ID of the AutoML Orchestrator pipeline.

By creating a MLMD lineage, we managed to track the status of the entire AutoML experiment, by starting from the run ID of the AutoML Orchestrator.

### 10.4.1 Kale Artifacts

In this section, we will explore the different types of Artifacts that Kale creates in order to create a MLMD lineage throughout the AutoML experiment.

To be able to submit and update different types of MLMD Artifacts, we created the `kale.common.artifacts` module which contains class definitions for all the Artifacts that we need. We will expose these classes in the following sections.

#### 10.4.1.1 The `MLMDArtifact` Base Class

This is a base class for objects that submit themselves in MLMD. Other classes that correspond to MLMD Artifacts and that expect their instances to be submitted in MLMD should subclass this class and they are expected to:

1. **inherit from the `MLMDArtifact` class**
2. **provide `artifact_type_name` and `artifact_property_types` attributes**
3. **implement the following properties: `artifact_properties` and `artifact_custom_properties`**
4. **provide an `artifact_uri` attribute, at run-time**
5. **over-write the `submit_artifact` method**

Listing 10.23: The `MLMDArtifact` base class for submitting Artifacts to MLMD

```

1 class MLMDArtifact(abc.ABC):
2     """A base class for objects submittable in MLMD,
3
4     Other classes that correspond to MLMD Artifacts and that expect their instances to be
5     submitted in MLMD should subclass this class.
6
7     Example:
8
9     >>> class MyClass(MLMDArtifact):
10         ...
11         artifact_type_name = "kale.MyClass"
12         artifact_property_types = {"prop1": metadata_store_pb2.STRING,
13                                   "prop2": metadata_store_pb2.INT}
14         ...
15         @property
16         def artifact_properties(self):
17             prop1 = metadata_store_pb2.Value(
18                 string_value=self.get_prop1())
19             prop2 = metadata_store_pb2.Value(
20                 int_value=self.compute_prop2())
21             return {"prop1": prop1, "prop2": prop2}
22         ...
23         @property
24         def artifact_custom_properties(self):
25             custom_prop = metadata_store_pb2.Value(

```

```

25     ...         string_value="custom_value")
26     ...         return {"custom_prop": custom_prop}
27     ...
28     >>> instance = MyClass()
29     >>> instance.artifact_uri = "gs://bucket/path/to/blob"
30     >>> artifact = instance.submit_artifact()
31
32     For more information on available property types, see
33     https://github.com/google/ml-metadata/blob/v0.29.0/ml\_metadata/proto/metadata\_store.
34     proto#L74-L81
35
36     Attributes:
37     artifact_type_name (str): The name of the ArtifactType
38     artifact_property_types (dict): The mapping of property names and their
39     MLMD types
40     artifact_uri (str): The URI of the Artifact
41
42     artifact_type_name: str = "kale.Artifact"
43     artifact_property_types: Dict = None
44     artifact_uri: str = None
45
46     @property
47     @abc.abstractmethod
48     def artifact_properties(self) -> Dict[str, metadata_store_pb2.Value]:
49         """Get the properties of the Artifact."""
50         pass
51
52     @property
53     def artifact_custom_properties(self) -> Dict[str,
54         metadata_store_pb2.Value]:
55         """Get the custom properties of the Artifact."""
56         return {}
57
58     def as_artifact(self) -> metadata_store_pb2.Artifact:
59         """Get the Artifact instance corresponding to this object.
60
61         Compose the Artifact instance based on all the information
62         held/computed by the object itself.
63
64         Returns:
65             metadata_store_pb2.Artifact: The generated Artifact
66         """
67     from ml_metadata.proto.metadata_store_pb2 import Artifact
68
69     log.info("Creating ArtifactType '%s'...", self.artifact_type_name)
70     artifact_type = mlmdutils.get_or_create_artifact_type(
71         type_name=self.artifact_type_name,

```

```

72     properties=self.artifact_property_types)
73     log.info("ArtifactType '%s' has ID %d", self.artifact_type_name,
74             artifact_type.id)
75
76     custom_properties = self.artifact_custom_properties.copy()
77     if hasattr(self, "_mlmd_list_index"):
78         custom_properties["list_index"] = metadata_store_pb2.Value(
79             int_value=self._mlmd_list_index)
80
81     return Artifact(uri=self.artifact_uri or "",
82                   type_id=artifact_type.id,
83                   properties=self.artifact_properties,
84                   custom_properties=custom_properties)
85
86     def submit_artifact(self) -> metadata_store_pb2.Artifact:
87         """Submit self to ML Metadata.
88
89         Returns:
90             metadata_store_pb2.Artifact: The submitted Artifact
91         """
92         artifact = self.as_artifact()
93
94         log.info("Creating '%s' Artifact...", self.artifact_type_name)
95         artifact.id = mlmdutils.put_artifact(artifact)
96         log.info("Successfully created '%s' Artifact with ID %d",
97                 self.artifact_type_name, artifact.id)
98         return artifact
99
100    def assign_list_index(self, list_index: int):
101        """Assign the artifact to a list of artifacts of the same type.
102
103        In a MLMD context, if some artifacts of type X is part of a list, then
104        all the artifacts of the same type belonging to the same context must
105        be part of the same of list (i.e. must have called
106        'assign_list_index')
107
108        Args:
109            list_index (int): The position of the artifact in the list
110        """
111        self._mlmd_list_index = list_index

```

#### 10.4.1.2 The *AutoMLConfiguration* Artifact

One of our goals, regarding the MLMD lineage of the AutoML experiment, was to be able to retrieve information about the suggested configurations that auto-sklearn provided. More specifically, we wanted to be able to retrieve this information just by using the run ID of the AutoML Orchestrator.



To implement that, we decided that when the *get-metalearning-configuration* step of the AutoML Orchestrator is executed, it should create AutoMLConfiguration Artifacts (subsection 10.2.2). This step also declares these Artifacts as outputs. This happens by submitting MLMD Events (8.6) that declare that the *AutoMLConfiguration* Artifacts are outputs of the Execution that corresponds to the *get-metalearning-configuration* step.

These AutoMLConfiguration Artifacts have a main property called *model\_data*. Before the *do\_run* method of the *GetMetaLearningConfigurations* step submits an AutoMLConfiguration Artifact on MLMD, it fills this property (10.2.2.4) with a dictionary that contains:

1. **The name of the model architecture of the suggested configuration.**
2. **The names and values of the parameters of that model.**

The following snippet shows the class definition, attributes and methods of the *AutoMLConfiguration* class.

Listing 10.24: The *AutoMLConfiguration* class for creating and submitting Artifacts that correspond to pipeline configurations

```

1  class AutoMLConfiguration(MLMDArtifact):
2      """An AutoML configuration Artifact."""
3      artifact_type_name = "kale.AutoMLConfiguration"
4      artifact_property_types = {"model_data": metadata_store_pb2.STRING}
5
6      def __init__(self, config_summary: Dict, run_id: str, estimator_name: str):
7          self.config_summary = config_summary
8          self.run_id = run_id
9          self.estimator_name = estimator_name
10
11     @property
12     def artifact_properties(self) -> Dict[str, metadata_store_pb2.Value]:
13         """Get kale.AutoMLConfiguration Artifact properties."""
14         model_data = metadata_store_pb2.Value(
15             string_value=json.dumps(self.config_summary))
16         return {"model_data": model_data}
17
18     @property
19     def artifact_custom_properties(self):
20         """Get kale.AutoMLConfiguration Artifact custom properties."""
21         run_id_prop = metadata_store_pb2.Value(string_value=self.run_id)
22         name_prop = metadata_store_pb2.Value(string_value=self.estimator_name)
23         return {"run_id": run_id_prop,
24             "name": name_prop}

```

### 10.4.1.3 Other Artifacts

Kale also creates and updates a number of other MLMD Artifacts, each one of which corresponds to a specific "entity" in the Kale-AutoML process that we built. Since they are not the main focus of the thesis, We will briefly expose these Artifacts and their purpose, in the following list:

- **Dataset:** An Artifact to track the features and targets of the ML dataset.
- **Model:** An Artifact to log the trained estimator that is produced during the *train-sklearn-estimator* step (10.3.3) of a Configuration Run.
- **Transformer:** An Artifact to log the fitted transformer that is produced during the *run-sklearn-transformer* step (10.3.2) of a Configuration Run.

## 10.4.2 The *AutoMLOExperiment* Object

This object is the main API that is used to track the status and the progress of an AutoML experiment. Given the run ID of the AutoML Orchestrator, an *AutoMLOExperiment* object fetches and displays all the information that denotes the state of the AutoML process.

Let us provide a code snippet with the class definition and the constructor method of the *AutoMLOExperiment* object:

Listing 10.25: The *AutoMLOExperiment* class

```

1 class AutoMLOExperiment():
2     """A status tracker for the AutoML process."""
3     def __init__(self, run_id: str):
4         self.automl_orchestrator_run_id = run_id
5         try:
6             kfputils.get_run(run_id)
7         except ApiException as e:
8             if e.status == 404:
9                 raise ValueError("Invalid run ID: '%s'. The run ID you"
10                                " provided does not correspond to a KFP run."
11                                % run_id)
12                 raise RuntimeError("Failed to retrieve KFP run with ID '%s': %s"
13                                    % (run_id, str(e)))
14
15         self.mlmd_client = mlmdutils.get_client()
16         self.context_type_name = "KfpRun"
17         self.context_name = mlmdutils.get_context_name_from_run_id(
18             self.automl_orchestrator_run_id)
19
20         self.context = None
21         self.automl_configuration_artifacts = []
22         self.automl_configuration_run_ids = []
23
24         self._update_all_info()

```

The constructor method of the *AutoMLOExperiment* initializes the object with just the **run ID of the AutoML Orchestrator**. Here is a list of the attributes of an *AutoMLOExperiment* object:

- **automl\_orchestrator\_run\_id** (str): The run ID of the AutoML Orchestrator.
- **context\_type\_name** (str): The MLMD *ContextType* name of the KFP run Context (8.6).

- **context\_name** (str): The name of the MLMD Context of the AutoML Orchestrator.
- **mlmd\_client** (ml\_metadata.MetadataStore): The MLMD client object to use for querying the MLMD database.
- **context** (ml\_metadata.proto.metadata\_store\_pb2.Context): The Context corresponding to the AutoML Orchestrator.
- **automl\_configuration\_artifacts** (list): A list of the AutoMLConfiguration Artifacts related to this experiment.
- **automl\_configuration\_run\_ids** (list): A list of the Configuration Run IDs.

To inform the user about the status of the AutoML process, the *AutoMLExperiment* object provides two main API methods:

1. **summary**: Informs the user about the status of the AutoML Orchestrator and Configurations Runs, and the metric scores of each Configuration Run.
2. **list\_configurations**: Informs the user about the estimator architecture and the parameters of each configuration suggested by auto-sklearn.

We will provide a more detailed view of these two API methods in the next sub-sections. But before that, we will expose the functionality of a critical utility method, called *\_update\_all\_info*:

Listing 10.26: The *\_update\_all\_info* utility method of the *AutoMLExperiment* class

```

1
2 def _update_all_info(self):
3     self.context = None
4     self.automl_configuration_artifacts = []
5     self.automl_configuration_run_ids = []
6
7     self.context = self.mlmd_client.get_context_by_type_and_name(
8         type_name=self.context_type_name, context_name=self.context_name)
9
10    if not self.context:
11        return
12
13    if "configuration_runs" in self.context.custom_properties:
14        config_run_ids_str = self.context.custom_properties[
15            "configuration_runs"].string_value
16        self.automl_configuration_run_ids = json.loads(config_run_ids_str)
17
18    self.automl_configuration_artifacts = (
19        mlmdutils.get_artifacts_by_context_and_type(
20            context_id=self.context.id,
21            type_name=AutoMLConfiguration.artifact_type_name,
22            sorted=True))

```

This method fetches all tracking information regarding the AutoML process at once. Doing this strives for consistency among the various information. More specifically, this method retrieves:

- the MLMD Context of the Orchestrator
- the AutoMLConfiguration Artifacts
- the Configuration Run IDs

### 10.4.2.1 The `list_configurations` Method

This API method shows information about the model architectures of the suggested configurations. It manages to do that by calling the `_update_all_info` method that we described earlier, in order to collect all the `AutoMLConfiguration` Artifacts (10.4.1.2) that were created by the `get-metalearning-configurations` step (10.2.2).

Let us view the code of the `list_configurations` method:

Listing 10.27: The `list_configurations` method

```

1 def list_configurations(self):
2     """Show information about the suggested model configurations."""
3     self._update_all_info()
4     if not self.automl_configuration_artifacts:
5         print("There are no configurations yet...\n")
6         return
7
8     ml.utils.print_suggested_models(self.automl_configuration_artifacts)

```

As we can see from above, `list_configurations` prints all the information related to the models of the pipeline configurations using the `print_suggested_models` utility function from Kale's `ml.utils` module:

Listing 10.28: The `print_suggested_models` function from the `kale.ml.utils` module

```

1 def print_suggested_models(artifacts: List[Artifact]):
2     """Print suggested models given an MLMD artifacts list."""
3     for idx, artifact in enumerate(artifacts, start=1):
4         print("Configuration", idx)
5         model_dict = json.loads(artifact.properties["model_data"].string_value)
6
7         if "name" not in model_dict:
8             raise RuntimeError("AutoMLConfiguration Artifact for"
9                               " Configuration%d doesn't provide a name for"
10                              " the estimator." % idx)
11         print("Estimator:", model_dict["name"])
12
13         if "parameters" not in model_dict:
14             raise RuntimeError("AutoMLConfiguration Artifact for"
15                               " Configuration%d doesn't provide any"
16                              " parameters." % idx)
17         print("Parameters:")
18
19         for param in model_dict["parameters"]:
20             print("    %s: %s" % (param, model_dict["parameters"][param]))
21         print("\n")

```

The following image is an example of how the `list_configurations` method outputs all the information regarding the models of the pipeline configurations when ran in a Jupyter Notebook cell.

```
[10]: automl_experiment.list_configurations()

Configuration 1
Estimator: lda
Parameters:
  shrinkage: manual
  tol: 0.0029112721339713465
  shrinkage_factor: 0.11778631707605758

Configuration 2
Estimator: libsvm_svc
Parameters:
  C: 68.60833575985336
  gamma: 0.03905068757955671
  kernel: rbf
  max_iter: -1
  shrinking: False
  tol: 0.02846559747114773

Configuration 3
Estimator: lda
Parameters:
  shrinkage: None
  tol: 0.09570561577075573

Configuration 4
Estimator: libsvm_svc
Parameters:
  C: 13.284077389890939
  gamma: 0.9060908941898536
  kernel: poly
  max_iter: -1
  shrinking: False
  tol: 0.06323853917206591
```

Figure 10.3: Example output of `list_configurations` method

### 10.4.2.2 The *summary* Method

The `summary` method is the main API method of the `AutoMLExperiment` object that enables the user to track the statuses of the AutoML Orchestrator and the Configuration Runs, as well as the metric scores that each Configuration Run produces.

To retrieve these information about the AutoML Orchestrator and the Configuration Runs, the `summary` method needs to retrieve their IDs. It already has acquired the Run ID of the AutoML Orchestrator (`automl_orchestrator_run_id` attribute). So, to retrieve the Configuration Run IDs, `summary` calls the `_update_all_info` utility method of the `AutoMLExperiment` class.

Listing 10.29: The `summary` API method of the `AutoMLExperiment` class

```

1 def summary(self):
2     """Show a summary of the AutoML process."""
3     # Retrieve the current status of the AutoML Orchestrator.
4     try:
5         status = kfputils.get_run(
6             self.automl_orchestrator_run_id).run.status
7     except ApiException as e:
8         if e.status == 404:
9             raise ValueError("Invalid run ID: '%s'. The run ID you"
10                              " provided does not correspond to a KFP run."
11                              % self.automl_orchestrator_run_id)
12         raise RuntimeError("Failed to retrieve KFP run with ID '%s': %s"
13                             % (self.automl_orchestrator_run_id, str(e)))
14     if not status:
15         print("The AutoML orchestrator has not started yet...")
16         return
17     print("AutoML orchestrator status: %s\n" % status)
18
19     self._update_all_info()
20     if not self.automl_configuration_artifacts:
21         print("There are no configurations yet...\n")
22         return
23
24     self._print_configuration_run_status_counts()
25     if self.automl_configuration_run_ids:
26         self._print_summary_table()

```

As it can be seen from the code snippet above, the *summary* method prints, serially, four different types of information:

1. **The status of the AutoML Orchestrator.** If the AutoML Orchestrator has not started yet, the method prints: *"The AutoML Orchestrator has not started yet..."* and exits its execution. Otherwise, it prints the status of the Orchestrator and continues with the items below.
2. **A counter for the Configuration Runs that have started.**
3. **A HTML table that counts the number of Configuration Runs that have a specific status.** To implement this, *summary* uses another method of the *AutoMLExperiment* class, called *\_print\_configuration\_run\_status\_counts*. The following snippet shows the code of the *\_print\_configuration\_run\_status\_counts* utility method:

Listing 10.30: The *\_print\_configuration\_run\_status\_counts* method of the *AutoMLExperiment* class

```

1 def _print_configuration_run_status_counts(self):
2     """Print the status of each Configuration Run."""
3     # Initially, print the number of started runs,
4     # out of all the suggested Configurations.
5     num_of_configs = len(self.automl_configuration_artifacts)

```

```

6     num_of_config_runs = len(self.automl_configuration_run_ids)
7
8     print("%d/%d Configuration runs have started.\n"
9           % (num_of_config_runs, num_of_configs))
10
11     # Then, print a table that shows how many runs
12     # are currently "Running", "Failed", "Succeeded" etc...
13     status_counts = {"Running": 0}
14     for status in kfputils.KFP_RUN_FINAL_STATES:
15         status_counts[status] = 0
16
17     for run_id in self.automl_configuration_run_ids:
18         try:
19             status = kfputils.get_run(run_id).run.status
20         except ApiException as e:
21             if e.status == 404:
22                 raise ValueError("Invalid run ID: '%s'."
23                                   " The run ID you"
24                                   " provided does not"
25                                   " correspond to a KFP"
26                                   " run." % run_id)
27             raise RuntimeError("Failed to retrieve KFP run"
28                               " with ID '%s':"
29                               " %s" % (run_id, str(e)))
30
31         if status not in status_counts:
32             status_counts[status] = 0
33             status_counts[status] += 1
34
35     headers = ["Status", "Count"]
36
37     if utils.is_ipython():
38         # If we are running in a Jupyter Notebook we display
39         # the table in HTML format.
40         from IPython.core.display import display, HTML
41         table = tabulate.tabulate(
42             tabular_data=list(status_counts.items()),
43             headers=headers,
44             tablefmt="html")
45         display(HTML(table))
46     else:
47         table = tabulate.tabulate(
48             tabular_data=list(status_counts.items()),
49             headers=headers)
50         print(table)
51     print("\n")

```

This method initially prints the number of started Configuration Runs out of all the

suggested pipeline configurations. Then, it prints a table that shows how many runs have a particular status, such as: **Running**, **Failed** or **Succeeded**. As in the case of the `run_automl` API function (10.1), this method is also meant to be executed inside a Jupyter Notebook, in KubeFlow's Jupyter Server environment (8.4.2). If that happens, then the table is pretty-printed in HTML format, using the `IPython.core.display` module [32]. Otherwise, it's printed in plain text format, using the `tabulate` Python package [33].

4. **A summary table that outputs the status and metric score of each Configuration Run.** To implement this, `summary` uses another method of the `AutoMLExperiment` class, called `_print_summary_table`. The following snippet shows the code of the `_print_summary_table` utility method:

Listing 10.31: The `_print_summary_table` method of the `AutoMLExperiment` class

```

1     def _print_summary_table(self):
2         """Print a summary table for the configuration runs."""
3         tabular_data = []
4
5         # This will be the header of the metrics column. When the first
6         # (metric_name, score) pair is received for a configuration run,
7         # the header will be changed to "Metric (<metric_name>)".
8         metric_header = "Metric"
9
10        for i, run_id in enumerate(self.automl_configuration_run_ids):
11            # Get the status of the configuration run that has started.
12            try:
13                status = kfputils.get_run(run_id).run.status
14            except ApiException as e:
15                if e.status == 404:
16                    raise ValueError("Invalid run ID: '%s'."
17                                    " The run ID you"
18                                    " provided does not"
19                                    " correspond to a KFP"
20                                    " run." % run_id)
21                raise RuntimeError("Failed to retrieve KFP run"
22                                    " with ID '%s':"
23                                    " %s" % (run_id, str(e)))
24            metric = "-"
25
26            # Get the MLMD Context of the configuration run, if exists.
27            context_name = mlmdutils.get_context_name_from_run_id(run_id)
28            context = self.mlmd_client.get_context_by_type_and_name(
29                type_name=mlmdutils.RUN_CONTEXT_TYPE_NAME,
30                context_name=context_name)
31
32            if not context:
33                tabular_data.append([i + 1, run_id, status, metric])
34            continue

```



```

35
36     # The Model's ArtifactType may not exist yet and be created
37     # during this AutoML experiment.
38     try:
39         artifacts = \
40             mlmdutils.get_artifacts_by_context_and_type(
41                 context_id=context.id,
42                 type_name=Model.artifact_type_name)
43     except NotFoundError:
44         artifacts = []
45
46     if not artifacts:
47         tabular_data.append([i + 1, run_id, status, metric])
48         continue
49
50     metric_dict = json.loads(
51         artifacts[0].properties["metrics"].string_value)
52     if not metric_dict:
53         tabular_data.append([i + 1, run_id, status, metric])
54         continue
55
56     metric_name = list(metric_dict.keys())[0]
57     metric = metric_dict[metric_name]
58     tabular_data.append([i + 1, run_id, status, metric])
59
60     # Change the header of the Metric column to also include the
61     # name of the metric.
62     metric_header = "Metric (%s)" % metric_name
63
64     headers = ["#", "KFP Run", "Status", metric_header]
65     client = kfputils._get_kfp_client()
66
67     # If we are running in a Jupyter Notebook we can turn
68     # the displayed Run IDs into clickable links.
69     if utils.is_ipython():
70         from IPython.core.display import display, HTML
71
72         # Get the table in HTML format.
73         table = tabulate.tabulate(tabular_data=tabular_data,
74                                 headers=headers,
75                                 tablefmt="html")
76
77         # Replace each configuration run ID string with HTML code that
78         # links to the KFP UI.
79         for run_id in self.automl_configuration_run_ids:
80             link = ("%s/#/runs/details/%s" %
81                   (client._get_url_prefix(), run_id))
82

```

```
83         html = ('<a href="%s" target="_blank">%s</a>'
84                % (link, run_id))
85
86         table = table.replace(run_id, html)
87
88         display(HTML(table))
89     else:
90         # Get the table in regular format.
91         table = tabulate.tabulate(tabular_data=tabular_data,
92                                  headers=headers)
93         print(table)
94     print("\n")
```

This method prints a summary table that shows the index, the run ID, the status and the metric score for each Configuration Run. Similarly with the case of `_print_configuration_run_status_counts`, if `_print_summary_table` is executed inside a Jupyter Notebook, in KubeFlow's Jupyter Server environment, then the table is pretty-printed in HTML format, using the `IPython.core.display` module [32].

Figure 10.4 is an example of how the `summary` method outputs the statuses of the AutoML Orchestrator and the Configuration Runs, as well as the metric scores that each Configuration Run produce.

What is also important about running the `summary` method inside a Jupyter Notebook, is the fact that each Configuration Run ID is a clickable link to the KFP UI page of the corresponding Configuration Run. Figure 10.5 shows the KFP UI page that pops up when clicking on the first Configuration Run of the previous example.

```
[15]: automl_experiment.summary()
AutoML orchestrator status: Running
4/4 Configuration runs have started.
```

Status	Count
Running	2
Succeeded	2
Skipped	0
Failed	0
Error	0

#	KFP Run	Status	Metric (accuracy)
1	9232db8b-a9d9-49d7-858a-efae9f05ccf4	Succeeded	0.9955555555555555
2	0119dd00-68c1-424d-ab1e-7be6df7bbc19	Succeeded	0.9911111111111112
3	4cf9f27f-d8b9-4541-999f-183a82caf927	Running	-
4	0b6c5de4-772c-45c2-9d30-8e6e607c7ecb	Running	-

Figure 10.4: Example output of `summary` method when executed in a Jupyter Notebook cell

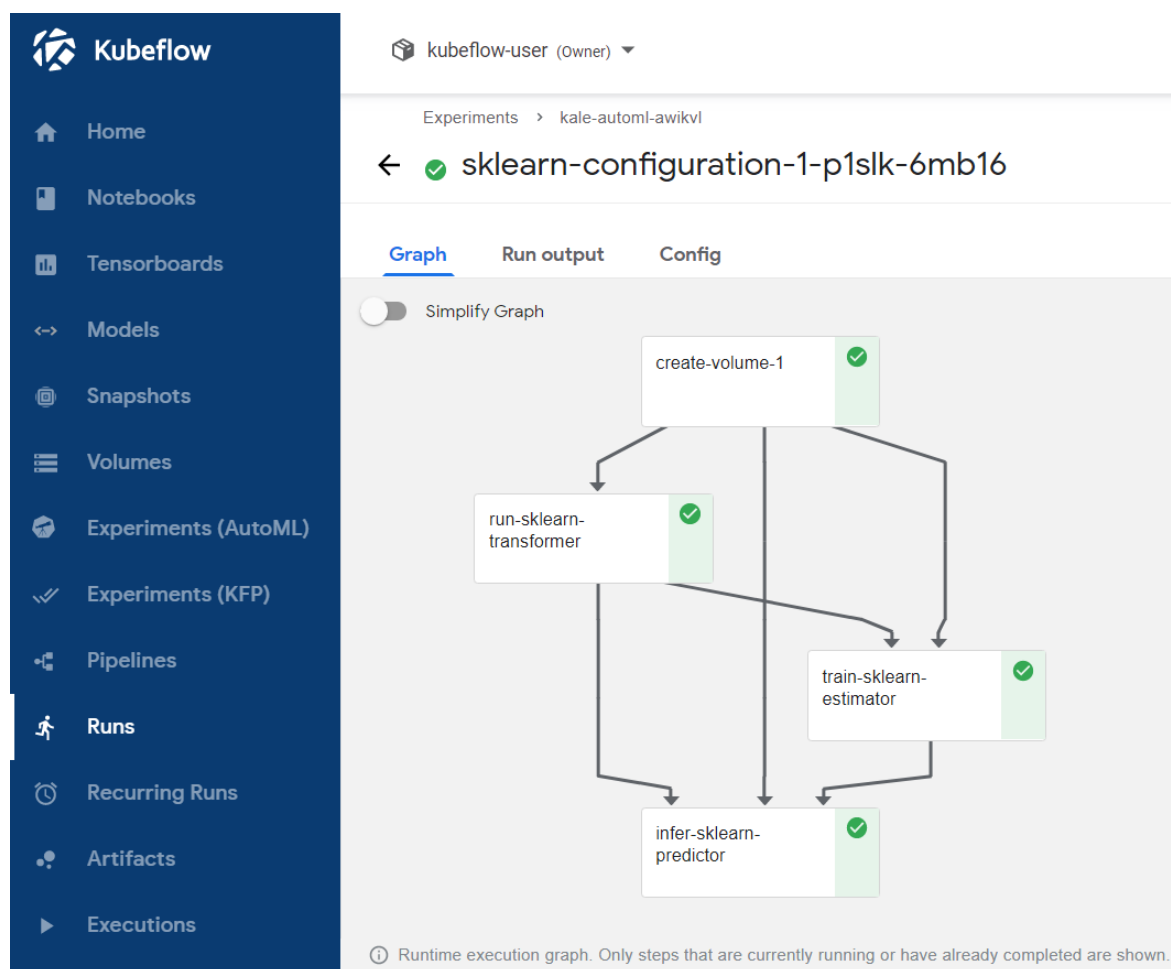


Figure 10.5: The KFP UI page for a Configuration Run

## Chapter **11**

# Evaluation

---

In this chapter, we will evaluate our mechanism’s performance and compare it against *auto-sklearn*’s latest version (currently *0.12.7*), which is the state of the art in the field of AutoML experiment libraries. More specifically, we measured the performance of our mechanism and of *auto-sklearn* on a number of well-known machine learning datasets. We tested our mechanism on datasets that are used both for regression and classification tasks, and vary in size.

For the evaluation of our mechanism, we compared ourselves against *auto-sklearn*’s mechanism **both with and without the ensemble construction mechanism** that *auto-sklearn* comes with. Experiments with ensemble construction, depending on the dataset, tend to need more time to produce a well-performing end model than the average Kale-AutoML experiment, which during our evaluations lasted about thirty minutes on average.

We were pleased to see that our mechanism performs exceptionally well for all datasets that we used, and **in some cases our mechanism beats auto-sklearn both with and without ensemble-model construction** enabled.

### 11.1 Tools, Methodology and Environment

We conducted our experiments in a Jupyter Lab environment of a MiniKF (8.4.5) installation, on a single node in Google Cloud Platform [34]. More specifically, the node came with a 16-core CPU (Intel Xeon 2.3 GHz - Broadwell) along with 60 GB of RAM. The Jupyter Server (8.4.2) that we used for running our experiments mounted 4 cores of the CPU along with 4 GB of RAM and 4GB of workspace volume, which we concluded were sufficient.

Our evaluation suite comprises 5 supervised learning datasets that we expose in table 11.1.

Each of our Kale-AutoML experiments was executed using our *run\_automl* (10.1) API function, and it used **five pipeline configurations** (*number\_of\_configurations=5*) while **at most two Configuration Runs** could run in parallel (*max\_parallel\_configurations=2*). Similarly, the Katib experiment that implements the hyper-parameter optimization part of our Kale-AutoML process ran **five trials** while **at most two Katib trials** could run in parallel.

For the corresponding *auto-sklearn* experiments, we used *auto-sklearn*’s *AutoSklearn*-

name	Machine Learning Task	Train/Test Ratio
MNIST (35)	multi-class classification	80/20
CIFAR-10 (36)	multi-class classification	80/20
GERMAN CREDIT DATA (37)	binary classification	75/25
WINE QUALITY (38)	regression	75/25
DIABETES (39)	regression	75/25

Table 11.1: The datasets that were used to evaluate our AutoML mechanism

*Classifier* and *AutoSklearnRegressor* API objects. The input parameter configurations that we provided to these objects are listed in table 11.2.

parameter	auto-sklearn	auto-sklearn + ensemble
time_left_for_this_task (sec)	2000	3000
per_run_time_limit (sec)	360	360
initial_configurations_via_metalearning	5	5
ensemble_size	0	5
ensemble_nbest	0	5
max_models_on_disc	50	50
seed	1	1
memory_limit (MB)	3072	3072

Table 11.2: The input parameters for *AutoSklearnClassifier* and *AutoSklearnRegressor* in our auto-sklearn experiments

## 11.2 Results

We show our measurements in table 11.3. We ran all the experiments sequentially and not in parallel in order to avoid any overuse of the CPU or the RAM. The underlined scores are the best ones out of the three experiment set-ups.

Dataset	Metric	auto-sklearn	auto-sklearn + ensemble	Kale-AutoML
MNIST	accuracy	0.885	0.942	<b>0.95</b>
CIFAR-10	accuracy	0.25	<u>0.31</u>	0.25
GERMAN CREDIT DATA	accuracy	0.776	<u>0.78</u>	<b>0.78</b>
WINE QUALITY	mean squared error	0.506	0.478	<b>0.457</b>
DIABETES	mean squared error	42.962	<u>42.539</u>	42.902

Table 11.3: Metric score measurements for all experiments.

We were pleased to see that in the cases of datasets *MNIST*, *GERMAN CREDIT DATA* and *WINE QUALITY* **our Kale-AutoML process scores equally high or even higher than auto-sklearn’s mechanism**. We attribute this improvement to the integration of our mechanism with Katib, which performs hyper-parameter optimization on our already trained highest-scoring model. Without the Katib integration, our mechanism has the same results as auto-sklearn without the ensemble construction technique.

We notice that, for the case of the *CIFAR-10* dataset, auto-sklearn's ensemble construction mechanism demonstrates a significant improvement in test scores in comparison with both the standard auto-sklearn meta-learning mechanism and our mechanism. *CIFAR-10* is the largest dataset in our suite, and this led to three out of five configuration runs in our AutoML experiment failing. The reason for the failure was that the cloned workspace volume that Kale mounted in each step of these Configuration Runs was not sufficient to marshal the pre-processed dataset that was produced by some of the *run-sklearn-transformer* steps (10.3.2).





## Chapter **12**

### Concluding Remarks

---

Our journey has finally reached its end. In this chapter, we will restate our contributions and summarize what our mechanism offers. Finally, we will close this thesis by mentioning future work that can be done to enrich our mechanism and bring it to its full potential.

#### **12.1 A Recap of our Mechanism**

We have designed, implemented and evaluated Kale's AutoML mechanism. Let us provide a recap of what it offers, once more:

- It starts AutoML experiments on Kubeflow with just a single API function call.
- It enable users to track the status of the experiment and the metric scores that it produces through a simple API object.
- It leverages Kubernetes' distributed nature and Kale's pipeline orchestration mechanism to distribute the parts of the experiment that can be parallelized.
- It uses *auto-sklearn*'s meta-learning mechanism to "warmstart" the process and Katib's hyper-parameter optimization mechanism to fine-tune the best performing model.
- It leverages Kale's data-versioning mechanism to make every single trained model easily reproducible and accessible even after the whole experiment is over.

## 12.2 Future Work

We can finally wrap this thesis up with the future research directions of our mechanism. We plan on pursuing these actively over the next months or years.

- Extend our *AutoMLOperiment* (10.4.2) API object to track the status of the Katib Experiment as well. Currently it only tracks the status of the AutoML Orchestrator and the Configuration Runs.
- Integrate auto-sklearn's ensemble construction mechanism to our Kale-AutoML mechanism to produce even stronger end-models.
- Extend the *Dataset* (10.1.1) class to include an evaluation set along with the train and test sets that it already includes.
- Enable users to limit the running time of each Configuration Run through Kale's *run\_automl* API, just like the case of the *auto-sklearn* API.

## Bibliography

---

- [1] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum και Frank Hutter. *Efficient and Robust Automated Machine Learning*. *Advances in Neural Information Processing Systems 28C*. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama και R. Garnett, επιμελητές, σελίδες 2962–2970. Curran Associates, Inc., 2015.
- [2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot και E. Duchesnay. *Scikit-learn: Machine Learning in Python*. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [3] *The official Kubeflow page*. <https://www.kubeflow.org/>. Accessed: 31-10-2021.
- [4] Johnu George, Ce Gao, Richard Liu, Hou Gang Liu, Yuan Tang, Ramdoot Pydipaty και Amit Kumar Saha. *A Scalable and Cloud-Native Hyperparameter Tuning System*, 2020.
- [5] *Kubeflow’s Documentation for Katib’s search algorithms*. <https://www.kubeflow.org/docs/components/katib/experiment/#search-algorithms-in-detail>. Accessed: 01-11-2021.
- [6] *Wikipedia article about meta-learning*. [https://en.wikipedia.org/wiki/Meta\\_learning\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Meta_learning_(computer_science)). Accessed: 01-11-2021.
- [7] *Kubernetes’ official web-page*. <https://kubernetes.io/>. Accessed: 30-10-2021.
- [8] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li και Alexander Smola. *AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data*. *arXiv preprint arXiv:2003.06505*, 2020.
- [9] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla και Carol Willing. *Jupyter Notebooks - a publishing format for reproducible computational workflows*. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Loizides και B. Schmidt, επιμελητές, σελίδες 87 – 90. IOS Press, 2016.
- [10] *KFP’s GitHub Repository*. <https://github.com/kubeflow/pipelines>. Accessed: 31-10-2021.

- [11] *The KALE project*. <https://github.com/kubeflow-kale/kale>. Accessed: 30-10-2021.
- [12] *Python's official page*. <https://www.python.org/>. Accessed: 31-10-2021.
- [13] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke και Travis E. Oliphant. *Array programming with NumPy*. *Nature*, 585(7825):357–362, 2020.
- [14] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, António H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt και SciPy 1.0 Contributors. *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*. *Nature Methods*, 17:261–272, 2020.
- [15] *Scikit-learn's Pipeline class*. <https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html>. Accessed: 31-10-2021.
- [16] Tejun Heo (Linux Kernel Developer). *Control Group v2*. <https://www.kernel.org/doc/Documentation/cgroup-v2.txt>. Accessed: 31-05-2021.
- [17] *DockerHub*. <https://hub.docker.com/>. Accessed: 03-11-2021.
- [18] *The YAML data-serialization language*. <https://en.wikipedia.org/wiki/YAML>. Accessed: 30-10-2021.
- [19] *The official etcd web-page*. <https://etcd.io/>. Accessed: 30-10-2021.
- [20] *The official TensorFlow page*. <https://www.tensorflow.org/>. Accessed: 31-10-2021.
- [21] *The official TensorFlow Extended (TFX) page*. <https://www.tensorflow.org/tfx/>. Accessed: 31-10-2021.
- [22] *The official website of Argo Project*. <https://argoproj.github.io/>. Accessed: 01-11-2021.
- [23] *Kubeflow's documentation for the MiniKF distribution*. <https://www.kubeflow.org/docs/distributions/minikf/>. Accessed: 31-10-2021.
- [24] *The official Rok page*. <https://www.arrikto.com/rok-data-management-platform/>. Accessed: 31-10-2021.

- [25] *Automating Jupyter Notebook Deployments to Kubeflow Pipelines with Kale*. <https://medium.com/kubeflow/automating-jupyter-notebook-deployments-to-kubeflow-pipelines-with-kale-a4ede38bea1f>. Accessed: 30-10-2021.
- [26] *Kale SDK documentation*. <https://docs.arrikto.com/user/kale/sdk/index.html>. Accessed: 30-10-2021.
- [27] *The DiGraph class for directed graphs with self loops of the networkx Python package*. <https://networkx.org/documentation/stable/reference/classes/digraph.html>. Accessed: 30-10-2021.
- [28] *MLMD's GitHub Repository*. <https://github.com/google/ml-metadata>. Accessed: 30-10-2021.
- [29] *TensorFlow's user-guide for MLMD entities*. <https://github.com/google/ml-metadata>. Accessed: 30-10-2021.
- [30] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchet, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney and Joaquin Vanschoren. *ASlib: A benchmark library for algorithm selection*. *Artificial Intelligence*, 237:41–58, 2016.
- [31] *Scipy's issparse function*. <https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.issparse.html#scipy.sparse.issparse>. Accessed: 11-10-2021.
- [32] *IPython's core.display module*. <https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.html>. Accessed: 29-10-2021.
- [33] *The tabulate Python package*. <https://github.com/astanin/python-tabulate>. Accessed: 29-10-2021.
- [34] *Google Cloud Platform*. <https://cloud.google.com/>. Accessed: 03-11-2021.
- [35] *The MNIST dataset*. <http://yann.lecun.com/exdb/mnist/>. Accessed: 03-11-2021.
- [36] *The CIFAR-10 dataset*. <https://www.cs.toronto.edu/~kriz/cifar.html>. Accessed: 03-11-2021.
- [37] *The German-Credit-Data dataset*. [https://archive.ics.uci.edu/ml/datasets/statlog+\(german+credit+data\)](https://archive.ics.uci.edu/ml/datasets/statlog+(german+credit+data)). Accessed: 03-11-2021.
- [38] *The Wine Quality dataset*. <https://archive.ics.uci.edu/ml/datasets/wine+quality>. Accessed: 03-11-2021.
- [39] *The Diabetes dataset*. <https://archive.ics.uci.edu/ml/datasets/diabetes>. Accessed: 03-11-2021.