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

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

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

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Ιωάννης Κωνσταντίνου
Καθηγητής ΕΜΠ	Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΠΘ

Αθήνα, Νοέμβριος 2021

.....

Κωνσταντίνος Χ. Ανδριόπουλος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

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 Kubeflow, το οποίο τρέχει πάνω στον Κυβερνήτη, και είναι μια τελευταίας τεχνολογίας πλατφόρμα για ενορχήστρωση ροών εργασίας μηχανικής μάθησης. Εκεί, εκμεταλλευτήκαμε τα οφέλη της κατανεμημένης φύσης του Κυβερνήτη, κατανέμοντας στο ¨νέφος¨ τα βήματα της διαδικασίας AutoML που μπορούσαν να τρέξουν παράλληλα.

Λέξεις Κλειδιά

Μηχανική Μάθηση, Αυτοματοποιημένη Μηχανική Μάθηση, Κυβερνήτης, 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 παντοτινά

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 εξαίρετων 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

List of Illustrations

List of Tables

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 κυκλοφορίας 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 πρ-οεπεξεργαστεί 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 μηχάνημα 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 ενορχήστρωση 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 σετ τιμών υπερ-παραμέτρων.

Το Katib υποστηρίζει έναν αριθμό αλγορίθμων αναζήτησης τους οποίους χρησιμοποιεί για να βρει σετ 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 σετ υπερ-παραμέτρων. **Η επιλογή του σωστού μοντέλου για ένα συγκεκριμένο σετ 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 σετ 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.

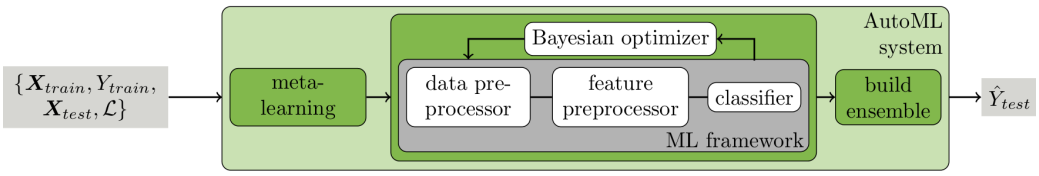

Figure 1.1: Το Auto-sklearn σε μία εικόνα

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 σετ 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 εμπεριστατωμένη 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 προηγμένη 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 κατανεμημένων 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 κατανεμημένη 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 διοχετεύσεις 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 διοχέτευση 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 χρήστη 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: περιèктес (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 μηχάνημα. 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 `mn mh` περιορίζει τη χρήση μνήμης και ο `cpuacct` 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 τέτοιον τρόπο που διαφορετικά σετ διεργασιών έχουν πρόσβαση σε διαφορετικά σετ πόρων. Μερικά παραδείγματα τέτοιων πόρων είναι οι `tautìthtec diergasi¸n`, τα `onìmata arqe₩wn` και αρχεία σχετιζόμενα με την πρόσβαση στο `d₫ktuo`.

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 υπαρχόντων 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 τετριμμένη 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 των 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 ενορχήστρωση από άκρη σε άκρη, επιτρέποντας και απλουστεύοντας την ενορχήστρωση 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 υποβάλει 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 απεικόνηση 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 ακρίβειας 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 κατανεμημένη 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 επισημειώνοντας κελιά κώδικα και κάνοντας κλικ σε ένα κουμπί διανομής στο επεκτεταμένο Jupyter UI. Το Kale είναι υπεύθυνο για τη μετατροπή του επισημειωμένου Notebook του χρήστη σε ένα λειτουργικό KubeFlow Pipeline, όπως και για να φροντίσει για τη διαβίβαση δεδομένων μεταξύ των βημάτων και τη διαχείρηση 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 επίγνωση 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

```
@step(name="my_step")
def step_1(in_1, in_2):
    # 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

```
class ExampleStep(Step):
    name = "example-step"
    def do_run(self, param1, param2):
        # 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

```
    def dsl(param="Hello"):
        my_out = step_1(param)
        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

```
    def dsl(param1="no", param2="no"):
        res1 = step_1(param)
        if param == "yes":
            res2 = step_2(res1)
            if param2 == "yes":
                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. MetaglwttÐ ei to antikeÐmeno Pipeline se èna Work ow YAML .
2. Dhmiou geÐ èna KubeFlow Pipeline meta o t¸nontac to Work ow sto KFP.

2.5.7 Mhqanismìc Dia Ð ashc Dedomènwn tou Kale

OÔtwc ¸ste na dia i ˆsei dedomèna metaxÔ hmˆtwn , to Kale p omh eÔei autìmata èna nèo PersistentVolume q hsimopoieÐ ènan upˆ qonta logikì dÐsko tou q¸ ou e gasÐac pou episunˆptetai se kˆ e pe ièkth tou matoc mÐac dioqèteushc .

To Kale p os èxei k¸dika sto tèloc thc ektèleshc thc sunˆ thshc do_run enìc matoc ¸ste na apojhkeÔsei ta antikeÐmena exìdou tou matoc se autì to koinì PersistentVol• umeClaim katˆ thn ektèlesh . Parìmoia , p os èxei k¸dika sthn a q thc ektèleshc enìc matoc , ¸ste na ort¸sei ta antikeÐmena exìdou twn prohgoÔmenwn hmˆtwn kai na ta d¸sei wc eisìdouc sto t èqwn ma .

Gia autì to skopì , to Kale qrhsimopoieÐ to domostoiqeÐo tou marshal , kai pio sug- kekrimèna tic mejìdouc save kai load tou domostoiqeÐou pou qrhsimopoioÔntai gia na apojhkeÔsei kai na ort¸sei ta dedomèna antÐstoiqa .

2.5.8 Ekdìseic kai Stigmiìtupa Dedomènwn sto Kale

Sthn pe Ðptwsh pou to Kale ekteleÐtai se ènan Notebook Server entìc enìc antikeimènou MiniKF (2.4.5), q hsimopoieÐ ton pelˆth Rok (2.4.5.1) gia:

1. Na anagnwrÐsei upˆrqontec logikoÔc dÐskouc q¸ ou ergasÐac / dedomènwn stonNote• book Server, na lˆ ei stigmiìtupˆ touc kai na touc prosart sei sta mata thc dioqèteushc . Me autìn ton t ìpo , o q¸ oc ergasÐac tou q sth (pou endeqomènwc perièqei arqeÐa dedomènwn egkatesthmènec exart seic) diathreÐtai sthn t èqousa dioqèteush .
2. Na lˆ ei stigmiìtupa logik¸n dÐskwn sto tèloc thc ektèleshc thc dioqèteushc , pros- èrontac ènan praktikì t ìpo anˆkthshc apojhkeumènwn antikeimènwn pou paˆq hkan katˆ thn ektèlesh thc dioqèteushc , wc antikeÐmena pl wc ekpaideumènwn mon- tèlwn epexergasmèna dedomèna .
3. Na lˆ ei stigmiìtupa logik¸n dÐskwn sthn a q ektèleshc kˆ e matoc , pros- èrontac ènan praktikì t ìpo anˆkthshc thc katˆstashc twn dedomènwn p in apì mÐa endeqìmenh apotuqÐa matoc.

2.6 Machine Learning Meta•Data

H Machine Learning Meta•Data (MLMD) [28] eÐnai mÐa ˆsh dedomènwn NoSQL ìpou katagˆ oume kai anaktoÔme metadedomèna sqeti ìmena me oèc e gasi¸n kai pei ˆmata mhqanik c mˆ hshc . Apì ed¸ kai sto ex c , a ana e ìmaste sth ˆsh Machine Learning Meta•Data wc "MLMD".

To MLMD o Ð ei ontìthtec pou mpo oÔn na apo hkeu oÔn sthn na anakth oÔn apì th ˆsh dedomènwn . Sth doleiˆ mac , sqetÐ oume kˆ e mÐa apì autèc tic ontìthtec me mÐa sugkek imènh idèa miac o c e gasi¸n enìc pei ˆmatoc mhqanik c mˆ hshc . Oi kÔ iec ontìthtec pou q hsimopoioÔme sth doleiˆ mac eÐnai :

- Contexts : 'Ena Context eÐnai mia ontìthta pou aposkopeÐ sto na empe ièqei mia omˆda ˆllwn ontot twn pou ìlec ma Ð o Ð oun èna genikìte o plaÐsio . AntistoiqoÔme autèc tic ontìthtec se KFP Runs.
- Executions : 'Ena Execution antistoiqeÐ se kˆti pou ekteleÐtai . 'Ena Execution mpo eÐ na mè oc enìc (pe issìte wn) Contexts . AntistoiqoÔme autèc tic ontìthtec se mata KFP.
- Artifacts : 'Ena Artifact eÐnai kˆti pou katanal¸netai parˆgetai apì èna Execu• tion . Epiplèon , mpo eÐ na mè oc enìc (perissìterwn) Contexts . Sth douleiˆ mac parˆgoume Artifacts pou antistoiqoÔn se ènan arijmì ontot twn sqetizìmenwn me th mhqanik mˆ hsh .
- Attributions : 'Ena Attribution dhl¸nei ìti èna Artifact eÐnai mè oc miac ontìthtac Context .
- Associations : 'Ena Association dhl¸nei ìti èna Execution eÐnai mè oc miac ontìthtac Context .
- Events : 'Ena Event dhl¸nei ìti èna Artifact eÐnai eÐsodoc èxodoc miac ontìthtac Execution .

Anat èxte stouc odhgoÔc q hst¸n tou MLMD tou TensorFlow [29] gia mia leptome epex ghsh twn ide¸n pÐsw apì autèc tic ontìthtec .

Figure 2.4: Mia Episkìphsh tou MLMD

Chapter 3

H Melèth mac pˆnw ston Mhqanismì Meta •Mˆ hshc tou Auto•sklearn

3.1 Episkìphsh

To pakèto auto•sklearn ([1]) eÐnai mia automatopoihmènh e galeio kh mhqanik c mˆ hshc pou apallˆssei èna q sth mhqanik c mˆ hshc apì thn epilog algo Ð mou kai th Ô mish upe pa amèt wn . Axiopoi¸ntac p ìs ata pleonekt mata sth Mpeô ian eltistopoÐhsh , th metamˆ hsh kai thn kataskeu enwmènwn montèlwn , epitugqˆnei na antika istˆ pl wc opoiond pote ektimht scikit•learn ([2]), gia epi lepìmenec e gasÐec mhqanik c mˆ hshc .

Gia touc skopoÔc aut c thc diplwmatik c ergasÐac , apomon¸same kai qrhsimopoi same ton mhqanismì metamˆ hshc tou auto•sklearn . Stic epìmenec enìhtec , a prosfèroume mÐa sÔnoyh thc melèthc mac pˆnw ston t ìpo leitourgÐac tou mhqanismoÔ kai mia leptomer perigraf twn epimè ouc tmhmˆtwn pou apartÐzoun ton mhqanismì .

Pa akˆtw , a ek èsoume mÐa a i mhmènh lÐsta hmˆtwn pou pe ig ˆ ei autìn ton mhqanismì metamˆ hshc . Ousiastikˆ , to auto•sklearn :

1. exˆgei èna set metaqa akth istik¸n apì èna sÔnolo dedomènwn eisìdou .

2. sugk Ðnei autì to diˆnusma metaqarakthristik¸n me ènan arijmì ˆllwn di- anusmˆtwn metaqarakthristik¸n pou eÐnai apojhkeumèna sth ˆsh dedomènwn metamˆjhs c tou kai Ðskei to pio parìmoio . Ousiastikˆ , kˆ e èna apì autˆ ta di- anÔsmata metaqarakthristik¸n antistoiqeÐ se èna sugkekrimèno set dedomènwn , ˆ a me ˆlla lìgia to auto•sklearn Ðskei to pio parìmoio set dedomènwn pou upˆrqei sth ˆsh dedomènwn metamˆjhs c tou .

3. p oteÐnei èna set diamorf¸sewn dioqèteushc mhqanik c mˆ hshc pou aj- molog jhke uyhlˆ sto pio parìmoio set dedomènwn . To auto•sklearn k atˆei epÐshc autèc tic ujmÐseic sth metˆ as tou , ma Ð me ta antÐstoiqa met ikˆ apotelèsmatˆ gia kˆ e set dedomènwn . Autèc oi proteinìmenec ujmÐseic eÐnai è aiec ìti a lei- tourgoÔn kalˆ sto set dedomènwn eisìdou .

3.2 Diamo ¸seic DioqeteÔsewn Mhqanik c Mˆ hshc

3.2.1 Episkìphsh

MÐa diamìrfwsh dioqèteushc mhqanik c mˆ hshc eÐnai ousiastikˆ mÐa leptomer c perigraf miac olìklh hc dioqèteushc mhqanik c mˆ hshc . To pakèto auto•sklearn perilambˆnei mÐa ˆsh dedomènwn pou perièqei mia plhj¸ra tètoiwn diamorf¸sewn . Ja perigrˆyoume aut th ˆsh dedomènwn metamˆ hshc se epìmenh enìthta .

O stìqoc tou auto•sklearn eÐnai na proteÐnei diamorf¸seic pou eÐnai è aiec na petÔqoun uyhl ajmologÐa gia èna dosmèno set dedomènwn kai ergasÐa mhqanik c mˆ hshc . To apìspasma k¸dika pa akˆtw deÐqnei èna pa ˆdeigma diamìrfwshc metamˆ hshc gia mia ergasÐa taxinìmhshc :

Listing 3.1: Mia diamì wsh metamˆ hshc gia taxinìmhsh

```
balancing :strategy , Value : 'weighting'
classifier :__choice __, Value : 'libsvm _svc'
classifier :libsvm _svc :C, Value : 6384.641073379224
classifier :libsvm _svc :coef0 , Value : •0.1592835134753816
classifier :libsvm _svc :degree , Value : 2
classifier :libsvm _svc :gamma, Value : 0.6866143858851854
classifier :libsvm _svc :kernel , Value : 'poly'
classifier :libsvm _svc :max_iter , Constant : •1
classifier :libsvm _svc :shrinking , Value : 'False'
classifier :libsvm _svc :tol , Value : 2.6500330000385803 e•05
data _preprocessing :categorical _transformer :categorical _encoding :__choice __, Value : 'no _encoding'
data _preprocessing :categorical _transformer :category _coalescence :__choice __, Value : '
    no_coalescense'
data _preprocessing :numerical _transformer :imputation :strategy , Value : 'median'
data _preprocessing :numerical _transformer :rescaling :__choice __, Value : 'normalize'
feature _preprocessor :__choice __, Value : 'no _preprocessing'
```

Pa ìmoia , to akìlou o apìspasma deÐqnei mia pragmatik diamìrfwsh metamˆ hshc pou a mpo oÔse na qrhsimopoihjeÐ gia mia ergasÐa palindrìmhshc :

Listing 3.2: Mia diamì wsh metamˆ hshc gia palind ìmhsh

```
data _preprocessing :categorical _transformer :categorical _encoding :__choice __, Value : 'no _encoding'
data _preprocessing :categorical _transformer :category _coalescence :__choice __, Value : '
    no_coalescense'
data _preprocessing :numerical _transformer :imputation :strategy , Value : 'most _frequent'
data _preprocessing :numerical _transformer :rescaling :__choice __, Value : 'robust _scaler'
data _preprocessing :numerical _transformer :rescaling :robust _scaler :q_max, Value :
    0.8280417820125114
data _preprocessing :numerical _transformer :rescaling :robust _scaler :q_min , Value :
    0.0781634653874277
feature _preprocessor :__choice __, Value : 'kitchen _sinks'
feature _preprocessor :kitchen _sinks :gamma, Value : 8.438432240830361 e•05
feature _preprocessor :kitchen _sinks :n_components , Value : 2984
regressor :__choice __, Value : 'ard _regression'
```

```
regressor :ard _regression :alpha _1, Value : 0.00044036509169446026
regressor :ard _regression :alpha _2, Value : 4.039147500668822 e•10
regressor :ard _regression :fit _intercept , Constant : 'True'
regressor :ard _regression :lambda _1, Value : 8.922721154590444 e•05
regressor :ard _regression :lambda _2, Value : 3.0105431227198885 e•05
regressor :ard _regression :n_iter , Constant : 300
regressor :ard _regression :threshold _lambda , Value : 1899.168836704701
regressor :ard _regression :tol , Value : 0.011611373742389547
```

'Opwc mpo eÐ kaneÐc na dei pa apˆnw , mia diamì wsh eÐnai eÐnai èna antikeÐmeno san lexikì pou o Ð ei èna ma p oepexe gasÐac kai kai èna ma ektimht gia mia dioqèteush mhqanik c mˆ hshc . Se epìmeno ke ˆlaio , a doÔme p¸c metat èpoume autèc tic pe ig a èc se antikeÐmena sklearn pou ìntwc ulopoioÔn p oepexe gastèc kai ektimhtèc .

3.2.2 O Q¸ oc Diamì wshc

O q¸ oc diamì wshc apoteleÐtai ousiastikˆ apì ìlec tic diamo ¸seic dioqeteÔsewn pou eÐnai ˆsimec gia mia dedomènh e gasÐa mhqanik c mˆ hshc . Gia pa ˆdeigma , sthn pe Ðptwsh miac e gasÐac taxinìmhshc , o q¸ oc diamì wshc a pe ièqei diamo ¸seic dioqeteÔsewn pou èqoun mìno taxinomhtèc san ektimhtèc . Pa ìmoia , sthn pe Ðptwsh miac e gasÐac palind ìmhshc , o q¸ oc diamì wshc a pe ièqei diamo ¸seic dioqeteÔsewn pou èqoun mìno palind omhtèc .

Wstìso , o tÔpoc thc dosmènhc e gasÐac mhqanik c mˆ hshc den eÐnai to mìno k it io pou eph eˆ ei th dom tou q¸ ou diamì wshc . Pio sugkek imèna , h dom tou q¸ ou diamì wshc eph eˆ etai apì t eic pa amèt ouc :

1. Ton tÔpo thc e gasÐac mhqanik c mˆ hshc .
2. Eˆn to set dedomènwn eisìdou eÐnai a aiì ìqi .
3. Eˆn to set dedomènwn eisìdou pe ièqei qamènec timèc ìqi .

Se epìmenh upoenìthta , a ana e oÔme sthn klˆsh XYDataManager , h opoÐa mac oh ˆei na upologÐsoume ìlec tic p oana e eÐsec pa amèt ouc . O q¸ oc diamì wshc eÐnai èna apì ta p ¸ta p ˆgmata pou upologÐ ei o mhqanismìc metamˆ hshc tou auto• sklearn , p otoÔ pa ˆxei tic p oteinìmenec diamo ¸seic dioqeteÔsewn .

3.2.2.1 H Sunˆ thsh get_con guration_space

To pakèto auto•sklearn ek ètei th sunˆ thsh get_con guration_space tou API h opoÐa pa ˆgei èna antikeÐmeno q¸ ou diamìrfwshc gia mia dedomènh ergasÐa mhqanik c mˆ hshc . O Ðste h upograf thc sunˆ thshc get_con guration_space tou API:

ˆ O Ðsmata Eisìdou :

info : 'Ena lexikì pou pe ièqei tic apa aÐthtec plh o o Ðec gia ton upologismì tou q¸ ou diamì wshc . Autì to lexikì p èpei na pe ièqei :

* task : 'Enan akè aio pou antistoiqeÐ ston tÔpo thc ergasÐac mhqanik c mˆ hshc .

* is_sparse : Mia tim Mpoul pou pe ig ˆ ei eˆn to set dedomènwn eisìdou eÐnai a aiì ìqi .
* has_missing : Mia tim Mpoul pou pe ig ˆ ei eˆn upˆ qoun qamènec ˆpei ec timèc sto set dedomènwn .

include_estimators : Mia lÐsta me ta onìmata twn ektimht¸n pou a qrhsimopoihjoÔn , apokleistikˆ . Aut tÐjetai apì proepilog wc None, opìte sumperilambˆnontai ìloi oi ektimhtèc , apì proepilog .

exclude_estimators : Mia lÐsta me ta onìmata twn ektimht¸n pou a exai e oÔn . Aut tÐ etai apì p oepilog wc None, opìte kanènac ektimht c den apokleÐetai .

include_preprocessors : Mia lÐsta me ta onìmata twn p oepexe gast¸n pou a q hsimopoih oÔn , apokleistikˆ . Aut tÐ etai apì p oepilog wc None, opìte sumpe ilam ˆnontai ìloi oi p oepexe gastèc , apì p oepilog .

exclude_preprocessors : Mia lÐsta me ta onìmata twn proepexergast¸n pou a exairejoÔn . Aut tÐjetai apì proepilog wc None, opìte kanènac proepexergast c den apokleÐetai , apì proepilog .

ˆ Tim Epist o c :

To antikeÐmeno q¸ ou diamì wshc

Se epìmenh enìthta , a mil soume gia to XYDataManager , èna antikeÐmeno klˆshc pou pa èqetai apì to auto•sklearn , to opoÐo upologÐ ei to lexikì info gia emˆc .

3.2.3 O Katˆlogoc Meta •Dedomènwn

O katˆlogoc metadedomènwn tou auto•sklearn eÐnai ènac p agmatikìc katˆlogoc entìc tou pakètou auto•sklearn ìpou eÐnai apo hkeumènh ìlh h meta •gn¸sh tou auto•sklearn . Aut h meta •gn¸sh o gan¸netai kat ' akolou Ða me to o mˆt ASlib ([30]). Autì shmaÐnei ìti ta metadedomèna kathgo iopoioÔntai se upokatalìgouc , anˆloga me t eic pa ˆgontec :

1. H met ik sunˆ thsh
2. O tÔpoc thc e gasÐac
3. H a aiìthta tou set dedomènwn

'Etsi , gia na d¸soume èna pa ˆdeigma , eˆn antastoÔme èna puknì set dedomènwn gia mia e gasÐa polutaxik c taxinìmhshc , me thn ak Ð eia wc met ikì , tìte autì a enèpipte ston upokatˆlogo accuracy_multiclass.classi cation_dense tou katalìgou metadedomènwn tou auto•sklearn .

Pa ìmoia , sthn pe Ðptwsh enìc a aioÔ set dedomènwn se mia e gasÐa palind ìmhshc , me to mèso apìluto s ˆlma wc met ik , o upokatˆlogoc eÐnai : mean_absolute_error_•regression_dense

3.2.4 H Klˆsh XYDataManager

H klˆsh XYDataManager pou pa èqetai apì to auto•sklearn exˆgei kai apo hkeÔei plh o o Đec pou eÐnai apa aÐthtec ¸ste na pa aq oÔn p oteinìmenec diamo ¸seic gia mÐa dosmènh e gasÐa mhqanik c mˆ hshc . Ed¸ eÐnai ta o Đsmata eisìdou gia ton kataskeuast XYDataManager :

- Xˆ: 'Enac pÐnakac pou pe ilam ˆnei ta deÐgmata tou set dedomènwn ekpaÐdeushc .
- yˆ: 'Enac pÐnakac pou pe ilam ˆnei touc stìqouc tou set dedomènwn ekpaÐdeushc .
- X_testˆ: 'Enac pÐnakac pou pe ilam ˆnei ta deÐgmata tou set dedomènwn dokim c .
- y_testˆ: 'Enac pÐnakac pou pe ilam ˆnei touc stìqouc tou set dedomènwn dokim c .
- taskˆ: 'Enac akè aioc pou antistoiqeÐ ston tÔpo thc e gasÐac mhqanik c mˆ hshc . Gia touc skopoÔc thc die gasÐac tou Kale•AutoML, autìc o akè aioc a eÐnai eÐte 1, eÐte2 eÐte4 pou antistoiqoÔn se palind ìmhsh , duadik taxinìmhsh kai polutaxik taxinìmhsh antÐstoiqa.
- feat_typesˆ: Mia lÐsta alfarijmhtik¸n pou perigrˆfei ton tÔpo kˆ e qarakthristikoÔ. H lÐsta èqei m koc Đso me ton arijmì twn qarakthristik¸n sto set dedomènwn . Kˆ e stoiqeÐo thc lÐstac mpo eÐ na eÐnai eÐte kathgorhmatikì eÐte arijmhtikì [1].
- dataset_nameˆ: To ìnoma tou set dedomènwn.

To akìlou o apìspasma k¸dika deÐqnei twn k¸dika thc sunˆ thshc kataskeuast tou XYDataManager :

Listing 3.3: H sunˆ thsh kataskeuast tou 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 ))
```

[1] H tim enìc kathgorhmatikoÔ qarakthristikoÔ dhl¸nei ìti èna deÐgma an kei se mia sugkekrimènh kathgorÐa. AutoÐ oi tÔpoi qarakthristik¸n èqoun èna peperasmèno arijmì upoy fiwn tim¸n . Ta arijmhtikˆ qarakthristikˆ apì thn ˆlh , ekfrˆzoun posotikˆ qarakthristikˆ enìc deÐgmatoc . 'Ena pa ˆdeigma kathgorhmatikoÔ qarakthristikoÔ eÐnai h q¸ a gènnhshc enìc atìmou , en¸ èna arijmhtikì qarakthristikì eÐnai to Ôyoc tou atìmou .

```
        else :
            self .info ['is _sparse' ] = 0
            self .info ['has _missing' ] = np.all (np.isfinite (X))

        label _num = {
            REGRESSION:1,
            BINARY_CLASSIFICATION: 2,
            MULTIOUTPUT_REGRESSION:y.shape [-1],
            MULTICLASS_CLASSIFICATION: len (np.unique (y)),
            MULTILABEL_CLASSIFICATION: y.shape [-1]
        }

        self .info ['label _num' ] = label _num[task ]

        self .data ['X _train' ] = X
        self .data ['Y _train' ] = y
        if X _test is not None :
            self .data ['X _test' ] = X_test
        if y _test is not None :
            self .data ['Y _test' ] = y_test

        if feat _type is not None :
            for feat in feat _type :
                allowed _types = ['numerical' , 'categorical' ]
                if feat .lower () not in allowed _types :
                    raise ValueError ("Entry '%s' in feat _type not in %s" %
                                      (feat .lower (), str (allowed _types )))

        self .feat _type = feat _type

        # TODO: try to guess task type!

        if len (y.shape ) > 2:
            raise ValueError ('y must not have more than two dimensions, '
                              'but has %d.' % len (y.shape ))

        if X .shape [0] != y.shape [0]:
            raise ValueError ('X and y must have the same number of '
                              'datapoints, but have %d and %d.' % (X.shape [0],
                                                                   y.shape [0]))
        if self .feat _type is None :
            self .feat _type = ['Numerical' ] * X.shape [1]
        if X .shape [1] != len (self .feat _type ):
            raise ValueError ('X and feat _type must have the same number of columns, '
                              'but are %d and %d.' %
                              (X.shape [1], len (self .feat _type )))
```

Katˆ thn a qikopoÐhs tou , èna antikeÐmenoXYDataManager exˆgei plh o o Đec pou

eÐnai apa aÐthtec gia thn die gasÐa metamˆ hshc kai tic apo hkeÔei sto pedÐo klˆshc info . Autì to pedÐo eÐnai ousiastikˆ èna lexikì Python pou pe ilam ˆnei ta akìlou a pedÐa :

- ˆ task : 'Enac akè aioc pou antistoiqeÐ ston tÔpo thc e gasÐac mhqanik c mˆ hshc .
- ˆ is_sparse : Mia tim Mpoul pou pe ig ˆ ei an to set eisìdou eÐnai a aiì ìqi . Gia na apo asÐsei ep ' autoÔ, to XYDataManager q hsimopoieÐ th sunˆ thsh issparse ([31]) to domostoiqeÐo sparse tou scipy ([14]).
- ˆ has_missing : Mia tim Mpoul pou pe ig ˆ ei eˆn upˆ qoun qamènec ˆpei ec timèc sto set dedomènwn.

'Ola ta pedÐa pou pe ig ˆyame pa apˆnw eÐnai apa aÐthta gia thn eÔ esh :

1. tou swstoÔ katalìgou metadedomènwn gia mia e gasÐa mhqanik c mˆ hshc . Se epìmenh enìthta , a pe ig ˆyoume p¸c to auto•sklearn qw Ð ei th meta• ˆsh tou se katalìgouc kai p¸c Ðskei to swstì katˆlogo metadedomènwn gia mÐa dedomènh e gasÐa mhqanik c mˆ hshc .
2. tou q¸ ou diamìrfwshc ston opoÐo a yˆxei to auto•sklearn gia upoy fiec di- amorf¸seic metamˆ hshc . Ja perigrˆyoume p¸c to auto•sklearn Ðskei ton q¸ o diamìrfwshc gia èna dedomèno peÐ ama se akìlou h enìthta .

3.2.5 Oi Klˆseic SimpleRegressionPipeline kai SimpleClassi cationPipeline

Autèc oi dÔo klˆseic ulopoioÔn thn e gasÐa taxinìmhshc . UlopoioÔn mÐa dioqèteush, pou pe ilam ˆnei mata p oepexe gasÐac kai èna ma ektimht sto tèloc .

'Ena antikeÐmeno aut¸n twn klˆsewn arqikopoieÐtai pern¸ntac èna antikeÐmeno diamìr- wshc metamˆ hshc (3.2) ston kataskeuast thc antÐstoiqhc klˆshc . Metˆ apì autì , mpo eÐ kaneÐc na kalèsei:

1. th mè odo t tou antikeimènou gia na e a mìsei thn dioqèteush se èna set dedomènwn ekpaÐdeushc.
2. th mè odo predict tou antikeimènou gia na kˆnei p o lèyeic se èna set dedomènwn dokim c .

3.3 Meta •Qa akth istikˆ

3.3.1 Episkìphsh

Se aut thn enìthta , a pe ig ˆyoume tic asikèc idèec tou upokeÐmenou mhqanismoÔ exagwg c metaqa akth istik¸n pou q hsimopoieÐ to auto•sklearn ([1]). Autì eÐnai èna apì ta asikˆ kommˆtia to pu na metamˆ hshc pou q hsimopoi same gia thn dianemhmènh AutoML die gasÐa mac.

To Auto-sklearn qw Ð ei to set twn metaqarakthristik¸n pou mpo eÐ na upologÐsei se dÔo kÔ iec kathgo Ðec :

1. Aplˆ Metaqa akth istikˆ : Autˆ ta metaqa akth istikˆ eÐnai upologistikˆ hnˆ , kai den apaitoÔn metasqhmatismoÔc sto set dedomènwn ¸ste na upologistoÔn .

2. 1HotEncoded Metaqarakthristikˆ : Autˆ ta metaqarakthristikˆ eÐnai pio ak iˆ upologistikˆ , kai upologÐzontai qrhsimopoi¸ntac th m tra qarakthristik¸n 1Ho• tEncoded tou set dedomènwn.

Ja analÔsoume autèc tic dÔo kÔ iec kathgorÐec metaqarakthristik¸n stic akìloujec up-oenìthtec .

3.3.2 Aplˆ Meta •Qa akth istikˆ

Autˆ ta metaqarakthristikˆ exˆgontai apeujeÐac apì to Dataset eisìdou , qw Ðc pro-hgoÔmenouc metasqhmatismoÔc proepexergasÐa , opìte eÐnai upologistikˆ hnˆ , genikˆ . Ed¸ eÐnai h pl hc lÐsta apl¸n meta •qa akth istik¸n pou mpo eÐ na upologÐsei to auto• sklearn :

- ˆ A i mìc antikeimènwn
- ˆ Loga i mikìc a i mìc antikeimènwn
- ˆ A i mìc klˆsewn
- ˆ A i mìc qa akth istik¸n
- ˆ Loga i mikìc a i mìc qa akth istik¸n
- ˆ A i mìc qa akth istik¸n me qamènec timèc
- ˆ Eˆn leÐpoun timèc ìqi
- ˆ A i mìc antikeimènwn me qamènec timèc
- ˆ Posostì antikeimènwn me qamènec timèc
- ˆ A i mìc qa akth istik¸n me qamènec timèc
- ˆ Posostì qa akth istik¸n me qamènec timèc
- ˆ A i mìc qamènwn tim¸n
- ˆ Posostì qamènwn tim¸n
- ˆ A i mìc a i mhtik¸n qa akth istik¸n
- ˆ A i mìc kathgo hmatik¸n qa akth istik¸n
- ˆ AnalogÐa a i mhtik¸n p oc kathgo hmatik¸n qa akth istik¸n
- ˆ AnalogÐa kathgo hmatik¸n p oc a i mhtik¸n qa akth istik¸n
- ˆ AnalogÐa qa akth istik¸n p oc antikeimènwn
- ˆ Loga i mik analogÐa qa akth istik¸n p oc antikeimènwn
- ˆ AnalogÐa antikeimènwn p oc qa akth istik¸n
- ˆ Loga i mik analogÐa antikeimènwn p oc qa akth istik¸n
- ˆ A i mìc em anÐsewn kˆ e klˆshc
- ˆ Elˆqisth pi anìthta klˆshc
- ˆ Mègisth pi anìthta klˆshc
- ˆ Mèsh tim thc pi anìthtac klˆshc
- ˆ Tupik apìklish thc pi anìthtac klˆshc

- ˆ A i mìc sum ìlwn [2]
- ˆ Elˆqistoc a i mìc sum ìlwn
- ˆ Mègistoc a i mìc sum ìlwn
- ˆ Tupik apìklish tou a i moÔ sum ìlwn
- ˆ 'A oisma twn a i m¸n sum ìlwn
- ˆ Ent opÐa klˆshc

3.3.3 Meta•Qa akth istikˆ 1HotEncoded

OÔtwc ¸ste na exˆgei autˆ ta metaqarakthristikˆ , to Dataset eisìdou upobˆlletai se èna metasqhmatismì . Pio sugkekrimèna , dhmiourgeÐtai mÐa m tra qarakthristik¸n 1HotEncoded . Autˆ ta metaqarakthristikˆ exˆgontai sthn pragmatikìthta apì aut th m tra qarakthristik¸n 1HotEncoded , ìqi apì to Ðdio to Dataset . Genikˆ, o upologis-mìc aut¸n twn qarakthristik¸n eÐnai upologistikˆ ak i ìc , a oÔ autˆ ta qarakthristikˆ pio prosanatolismèna p oc thn epist mh dedomènwn apì ta aplˆ metaqarakthristikˆ pou katagrˆyame parapˆnw . Ed¸ eÐnai h pl hc lÐsta twn meta •qa akth istik¸n 1HotEncoded pou mpo eÐ na upologÐsei to auto•sklearn:

- ˆ Loxìthtec
- ˆ Elˆqisth loxìthta
- ˆ Mègisth loxìthta
- ˆ Mèsh loxìthta
- ˆ Tupik apìklish loxìthtac
- ˆ Ku t¸seic
- ˆ Elˆqisth kÔ twsh
- ˆ Mègisth kÔ twsh
- ˆ Mèsh kÔ twsh
- ˆ Tupik apìklish kÔ twshc

T¸ a pou h gn¸sh mac twn tÔpwn metadedomènwn pou uposth Ð ei to auto•sklearn èqei epekta eÐ, eÐmaste ètoimoi na exe eun soume p¸c mpo eÐ kaneÐc na q hsimopoi sei to auto•sklearn ¸ste p ˆgmati na upologÐsei autˆ ta meta •qa akth istikˆ apì èna dosmèno set dedomènwn eisìdou.

[2] O ì oc sÔm olo ek ˆ ei thn tim pou mpo eÐ na èqei èna kathgo hmatikì qa akth istikì .

3.3.4 Suna t seic API gia Exagwg Meta •Qa akth istik¸n

Gia na upologÐsei ta meta •qa akth istikˆ pou an koun stic dÔo kathgorÐec pou analÔsame parapˆnw , to auto•sklearn prosfèrei dÔo kÔ iec sunart seic API:

1. calculate_all_metafeatures_with_labels
2. calculate_all_metafeatures_encoded_labels

Kai oi dÔo autèc suna t seic lam ˆnoun tic Ðdiec pa amèt ouc wc eisìdouc . Ac doÔme autèc tic pa amèt ouc leptome ¸c :

- X: Ta deÐgmata tou set dedomènwn ekpaÐdeushc
- y: Oi stìqoi tou set dedomènwn ekpaÐdeushc .
- categorical : Mia lÐsta Boolean tim¸n pou èqei m koc Ðso me ton arijmì qarak-thristik¸n se kˆe deÐgma . An èna stoiqeÐo sth lÐsta eÐnai True , tìte to antÐstoiqo qarakthristikì ewreÐtai wc kathgorhmatikì qarakthristikì . Eidˆllwc , eÐnai èna arijmhtikì qarakthristikì .
- dataset_name : To ìnoma tou set dedomènwn.
- dont_calculate : 'Ena set meta•qa akth istik¸n pou den a p èpei na upologistoÔn gia to sugkek imèno set dedomènwn eisìdou kai e gasÐa mhqanik c mˆ hshc .

H paˆmet oc dont_calculate pou ana è ame pa apˆnw q hsimopoieÐtai katˆ kÔ io lìgo gia na apokleÐsei meta •qa akth istikˆ stic pe ipt¸seic e gasi¸n palind ìmhshc . Pio sugkek imèna , e ìson oi e gasÐec palind ìmhshc den èqoun klˆseic wc stìqouc , ta akìlou a metaqa akth istikˆ p èpei na apokleistoÔn :

- A i mìc klˆsewn
- A i mìc em anÐsewn kˆe klˆshc
- Elˆqisth pi anìthta klˆshc
- Mègisth pi anìthta klˆshc
- Mèsh tim thc pi anìthtac klˆshc
- Tupik apìklish thc pi anìthtac klˆshc
- Ent opÐa klˆshc

Kai oi dÔo autèc suna t seic epist è oun èna antikeÐmeno DatasetMetafeatures apì to domostoiqeÐometalearning.metafeatures tou auto•sklearn . Autì to antikeÐmeno ousiastikˆ katˆ èna lexikì twn upologismènwn metaqa akth istik¸n sto pedÐo tou metafeature_• values .

3.3.5 H Klˆsh MetaBase

To antikeÐmeno klˆshc MetaBase eÐnai ènacontainer gia metadedomèna set dedomènwn (timèc metaqarakthristik¸n), diamorf¸seic dioqeteÔsewn kai apotelèsmata pei amˆtwn . EÐnai ousiastikˆ èna peritÔligma gÔ w apì th metagn¸sh tou auto•sklearn , pou apo-hkeÔetai ston katˆlogo metadedomènwn pou perigrˆyame prohgoumènwc .

Se èna antikeÐmenoMetaBase, to auto•sklearn apo hkeÔei ta metaqa akth istikˆ enìc set dedomènwn, ka¸c epÐshc kai ta apotelèsmata epikÔ wshc dia ì wn diamo¸sewn dioqeteÔsewn gia th sugkek imènh e gasÐa mhqanik c mˆ hshc .

Gia na kataskeuˆsoume èna antikeÐmeno MetaBase , p èpei na p omh eÔsoume :

- ˆ èna q¸ o diamì wshc
- ˆ ènan katˆlogo metadedomènwn

Epiplèon , a oÔ a qikopoi soume èna antikeÐmeno MetaBase mpo oÔme na p os èsoume kataq¸ hsh enìc set dedomènwn q hsimopoi¸ntac th mè odo add_dataset . Aut h mè odoc apaiteÐ:

- ˆ to ìnoma tou set dedomènwn
- ˆ èna antikeÐmeno DatasetMetafeatures pou pe ièqei tic timèc metaqa akth istik¸n gia to antÐstoiqo set dedomènwn

H p os kh enìc set dedomènwn eisìdou sto antikeÐmeno MetaBase eÐnai apa aÐthth gia ton upologismì twn proteinìmenwn diamorf¸sewn gia autì to set dedomènwn qrhsi-mopoi¸ntac th sunˆ thsh suggest_via_metalearning , thn opoÐa a analÔsoume se epìmenh enìthta .

3.4 H Sunˆ thsh suggest_via_metalearning

H sunˆ thsh suggest_via_metalearning tou domostoiqeÐou autosklearn.metalearning.mismbo eÐnai mÐa apì tic shmantikìterec sunart seic tou API sto mhqanismì metamˆ hshc tou auto•sklearn , a oÔ eÐnai aut pou pa ˆgei tic proteinìmenec diamorf¸seic dioqèteushc gia èna set dedomènwn eisìdou kai mÐa ergasÐa mhqanik c mˆ hshc . Ac doÔme pio analutikˆ thn upograf thc sunˆ thshc :

- ˆ O Ðsmata Eisìdou :

  meta_base : 'Ena antikeÐmenoMetaBase a qikopoihmèno me to swstì katˆlogo metadedomènwn kai to q¸ o diamì wshc ston opoÐo h suggest_via_metalearning a ana ht sei gia thn p oteinìmenec diamo¸seic .

  dataset_name : To ìnoma tou set dedomènwn gia to opoÐo a qrhsimopoihjoÔn oi proteinìmenec diamorf¸seic . To set dedomènwn p èpei na prosteje sto an-tikeÐmenoMetaBase. Autì mpo eÐ na gÐnei me th mè odo MetaBase.add_dataset pou epideÐxame sthn upoenìthta subsection 3.3.5.

  metric : H met ik sunˆ thsh .

  task : 'Enac akè aioc pou antistoiqeÐ ston tÔpo thc ergasÐac mhqanik c mˆ hshc .

  sparse : MÐaMpoul pou pe igˆ ei eˆn to set dedomènwn eisìdou eÐnai a aiì ìqi .

  num_initial_con gurations : O a i mìc twn p oteinìmenwn diamo¸sewn pou a pa aq oÔn .

- ˆ Tim Epist o c :

con gurations : Mia lÐsta me tic p oteinìmenec diamo ¸seic .

Se èna akìlou o kefˆlaio , a perigrˆyoume p¸c opÐs io tm ma mhqanik c mˆ hshc tou Kale qrhsimopoieÐ thn suggest_via_metalearning gia na pa ˆxei tic proteinìmenec diamorf¸seic pou a gìte a a metatrapoÔn se pragmatikèc dioqeteÔseic KFP.

3.5 Boh htikèc Suna t seic kai Klˆseic

3.5.1 H Klˆsh InputValidator

H InputValidator pou pa èqetai apì to domostoiqeÐo autosklearn.data.validation eÐnai mÐa q hstik klˆsh pou mpo eÐ na q hsimopoih eÐ gia na e ai¸sei ìti to set dedomènwn eisìdou summo ¸netai stic apait seic tou auto•sklearn . H asik mè odoc tou API thc InputValidator onomˆ etai validate kai h upog a thc pa ousiˆ etai pa akˆtw :

ˆ O Ðsmata Eisìdou :

X : Ta deÐgmata enìc set dedomènwn
y : Oi stìqoi enìc set dedomènwn .
is_classi cation : MÐa boolean tim pou ek ˆ ei eˆn h e gasÐa gia thn opoÐa a q hsimopoih eÐ to set dedomènwn eisìdou eÐnai taxinìmhsh ìqi .

ˆ Timèc Epist o c :

X : Ta epiku wmèna deÐgmata.
y : Oi epiku wmènoi stìqoi .

H InputValidator.validate ousiastikˆ ulopoieÐ dÔo leitou gikìthtec :

1. Elègqei ìti o a i mìc deigmˆtwn antistoiqeÐ ston a i mì stìqwn sto set dedomènwn .
2. Elègqei ìti to set dedomènwn apoteleÐtai mìno apì a i mhtikˆ dedomèna .
3. Anˆloga me ton tÔpo thc ergasÐac , pou ekfrˆzetai apì to ì isma eisìdou is_classi • cation , elègqei ìti oi stìqoi tou set dedomènwn mpo oÔn na qrhsimopoihjoÔn gia aut thn ergasÐa (taxinìmhsh palindrìmhsh).

Chapter 4

H P osèggish mac

'Opwc exhg same se p ohgoÔmeno ke ˆlaio , ta pei ˆmata AutoML pe ièqoun mata ta opoÐa mpo oÔn na pa allhlopoih oÔn . Se ènan topikì upologist , mìlic o mhqanismìc meta•mˆ hshc tou auto•sklearn upologÐsei tic p oteinìmenec diamo ¸seic dioqèteushc , to auto•sklearn ekpaideÔei tic antÐstoiqec dioqeteÔseic mhqanik c mˆ hshc topikˆ , san pa ˆllhlec die gasÐec , ètsi ¸ste h dioqèteush pou me thn kalÔte h apìdosh na e eÐ kai na epist a eÐ ston q sth . O stìqoc mac tan na meta è oume olìklh h aut thn diadikasÐa ston Ku e n th , ekmetalleuìmenoi ton mhqanismì eno q st wshc dioqeteÔsewn mhqanik c mˆ hshc tou Kale, ètsi ¸ste na ekpaideÔoume montèla mhqanik c mˆ hshc wc pa ˆllhlec dioqeteÔseic sto Kube ow. Dhmiou g same ènan mhqanismì pou epit èpei thn ektèlesh pei amˆtwn AutoML apodotikˆ , kai katanemhmèna , ston Ku e n th .
Ac pe ig ˆyoume ta mata thc die gasÐac tou Kale gia pei ˆmata AutoML:

1. O q sthc pa èqei èna sÔnolo dedomènwn kai ton tÔpo thc e gasÐac mhqanik c mˆ hshc (kathgo iopoÐhsh palind ìmhshc) wc eÐsodo sthn sunˆ thsh run_automl() tou Kale.
2. To Kale dhmiou geÐ mÐa dioqèteush tou Kube ow . H dioqèteush aut onomˆ etai Eno qhst wt c kai a ana e ìmaste se aut me autì to ìnoma gia to upìloipo tou ke alaÐou .
3. To Kale epist è ei èna antikeÐmeno AutoMLExperiment ston q sth (tim epist o c thc sunˆ thshc run_automl()). To antikeÐmeno autì ousiastikˆ a eÐnai èna ergaleÐo pa akoloÔ hshc thc katˆstashc olìklh hc thc diadikasÐac AutoML.
4. Q hsimopoi¸ntac ton mhqanismì meta •mˆ hshc tou auto•sklearn, o Enorqhstrwt c upologÐzei mÐa lÐsta me proteinìmenec diamorf¸seic dioqeteÔsewn mhqanik c mˆ hshc gia to sÔnolo dedomènwn kai tÔpo ergasÐac mhqanik c mˆ hshc pou o q sthc èdwse wc eÐsodo (sthn sunˆ thsh run_automl). Kˆ e mia apì autèc tic di-amorf¸seic ousiastikˆ perigrˆfei mÐa olìklh h dioqèteush mhqanik c mˆ hshc .
5. Gia kˆ e proteinìmenh diamìrfwsh dioqèteushc , o Enorqhstrwt c dhmiou geÐ mia nèa dioqèteush tou Kube ow . Oi dioqeteÔseic autèc onomˆzontai Roèc Diamìr-wshc kai a anaferìmaste se autèc qrhsimopoi¸ntac autì to ìnoma gia to upìloipo tou kefalaÐou . Kˆ e mÐa apì autèc tic Roèc Diamìrfwshc ulopoieÐ thn dioqèteush mhqanik c mˆ hshc pou h antÐstoiqh diamìrfwsh dioqèteushc perigrˆfei .

6. Oi Roèc Diamìrfwshc t èqoun pa ˆllhla , pern¸ntac to sÔnolo dedomènwn apì èna st ˆdio p o •epexe gasÐac, ekpaideÔontac to montèlo , kai par ˆgontac apotelès-mata dokimastikoÔ set en¸ o Enorqhstrwt c elègqei thn exèlixh touc .
7. Mìlic ìlec oi Roèc Diamìrfwshc oloklhr¸soun thn leitourgÐa touc , o Enorqh-strwt c sugkentr¸nei ta apotelèsmata touc kai me ˆsh aut ˆ epilègei thn kalÔte h Ro Diamìrfwshc .
8. O Enorqhstrwt c dhmiou geÐ èna peÐ ama Katib ètsi ¸ste na eltistopoih-eÐ pe aitè w to ekpaideumèno montèlo thc Ro c Diamìrfwshc me to megalÔte o sko .
9. To Kale apojhkeÔei to ekpaideumèno kai eltistopoihmèno montèlo kai tra-ˆei èna pl wc anaparag¸gimo stigmiìtupo Rok (2.4.5.1) tou logikoÔ dÐskou pou to perièqei èts ¸ste o q sthc na mpo eÐ na èqei prìsbash sto montèlo autì a gìte a .

To Kale trabˆei stigmiìtupa logik¸n dÐskwn , ìqi mìno sto tèloc , all ˆ se k ˆ e ma tou Enorqhstrwt kai twn Ro¸n Diamìrfwshc parèqontac èna olikì t ìpo na anakthjoÔn ta apojhkeumèna ekpaideumèna montèla pou pa ˆq hkan kat ˆ thn di ˆ keia thc diergasÐac AutoML.

Aut h diplwmatik e gasÐa esti ˆ ei ku Ðwc sta mata apì 3 èwc kai 7 thc die gasÐac AutoML tou Kale . Pa ìla aut ˆ , a pa ˆsqoume mia epa k an ˆlush thc leitou gikìthtac kai tou upoloÐpou mhqanismoÔ .

4.1 H Sunˆ thsh run_automl

Se aut thn enìthta , a perigrˆyoume thn mo thc sunˆ thshc diepa c tou Kale gia peirˆmata AutoML. Ousiastikˆ , oi q stec a t èqoun peirˆmata AutoML sto Kube ow, me mia mìno kl sh sunˆ thshc Python. Aut h sunˆ thsh , ìpwc ana è ame prohgoumènwc , onomˆzetai run_automl kai eÐnai ousiastikˆ èna shmeÐo eisìdou gia ton mhqanismì AutoML.

Listing 4.1: H sunˆ thsh diepa c run_automl gia dhmiou gÐa pei amˆtwn AutoML me to Kale

```
def run _automl (
    dataset : Dataset , task : MLTask, metric : Callable ,
    number_of _configurations : int = 5,
    max_parallel _configurations : int = 3,
    tuner : Optional [katib .V1beta1ExperimentSpec ] = None
) -> AutoMLExperiment :
    """Runs an AutoML pipeline to find the best model for the input dataset.

    [... Explain how the AutoML process works ...]

    Args:
        dataset (common.artifacts.Dataset): The input dataset for the ML task
        task (types.MLTask): One of kale.ml.Task
        metric (Callable): A callable object with the following call signature

            >>> class my _metric:
            >>>     def __call __(self, target, x _test):
            >>>         return self. _compute _my_metric _value(target, x _test)

            The name of the logged metric will be `metric.name`, if the object
            has such attribute. Otherwise ``metric. __name__ ``.
            (Auto)SKLearn metrics are supported, example:

            >>> from autosklearn.metrics import accuracy

            To log a different metric name from the input function name
            (``foo. __name__ ``), do:

            >>> foo.name = "<custom _name>"

        number_of _configurations (int): The N-best configurations to run
            (defaults to 5)
        max_parallel _configurations (int): The maximum number of Configuration
            Runs to run in parallel (defaults to 3)
        tuner (katib.V1beta1ExperimentSpec): Provide a Katib spec to run HP
            Tuning over the best performing configuration.

            Cannot set algorithm and parameters. To set objective
            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 )
```

Oi q stec mpo oÔn na eisˆgoun thn sunˆ thsh run _automl() me mÐa apl entol
import : » from kale.ml import run _automl

Ac d¸soume mia leptome ex ghsh twn pa amèt wn eisìdou thc run _automl() :

ˆ dataset : To sÔnolo dedomènwn eisìdou gia to peÐ ama AutoML. Autì p èpei na eÐnai èna antikeÐmenoDataset tou Kale pou na pe ièqei olìklh o to sÔnolo dedomènwn mhqanik c mˆ hshc tou q sth .

ˆ task : O tÔpoc thc e gasÐac epopteuìmenhc mhqanik c mˆ hshc . Autì mpo eÐ na eÐnai èna apì ta pa akˆtw :

1. duadik kathgo iopoÐhsh
2. kathgo iopoÐhsh pollapl¸n tˆxewn
3. apl opis od ìmhsh

ˆ metric : Mia klht sunˆ thsh met ik c h opoÐa a qrhsimopoihjeÐ gia na axiologheÐ to ekpaideumèno montèlo pou a paraqjeÐ apì thn kˆ e Ro Diamìrfwshc .

ˆ number_of_con gurations : O a i mìc twn p oteinìmenwn diamo ¸sewn pou a exˆgei o Eno qhst wt c

ˆ max_parallel_con gurations : O mègistoc a i mìc twn Ro¸n Diamì wshc pou a t èqoun pa ˆllhla .

ˆ tuner : 'Ena antikeÐmenospec enìc pei ˆmatoc Katib to opoÐo a q hsimopoih eÐ gia pe aitè w eltistopoÐhsh pa amèt wn tou kalÔte ou montèlou .

'Opwc ana è ame sthn a q autoÔ tou kefalaÐou , h sunˆ thsh diepa c run_automl() epist è ei èna antikeÐmeno AutoMLExperiment (4.4.2). Autì to antikeÐmeno , kai ousiastikˆ oi mè odoi tou , a epitrèyoun ston q sth na parakoloujeÐ thn katˆstash tou peirˆmatoc . Ja anaptÔxoume perissìtero thn pa akoloÔ hsh thc katˆstashc tou peirˆmatoc sthn enìthta 4.4, a oÔ p ¸ta èqoume analÔsei thn leitourgikìthta twn dioqeteÔsewn tou Enorqhstrwt (4.2) kai twn Ro¸n Diamìrfwshc (4.3).

Pa akˆtw paratÐjetai mia arijmhmènh lÐsta me mata pou perigrˆfei pwc h sunˆ thsh diepa c run_automl() kata è nei na purodot sei olìklh h thn diadikasÐa AutoML:

1. Q hsimopoieÐ ton apo hkeutikì mhqanismì tou Kale gia na apo hkeÔsei tic pa amèt ouc eisìdou thc ston logikì dÐsko tou Pod kai Ôste a t a ˆei èna stigmiìtupo Rok tou dÐskou èstsi ¸ste ta mata tou Eno qhst wt kai twn Ro¸n Diamì wshc mpo oÔn na p osa t soun ton klwnopoihmèno dÐsko gia na oun kai na o t¸soun tic pa amèt ouc .
2. Kataskeuˆ ei èna antikeÐmeno kale.PipelineCon g object me ìlec tic asikèc pe ig a ikèc plh o o Ðec (p.q: to ìnoma) tou Eno qhst wt .
3. Kataskeuˆ ei èna antikeÐmeno kale.processors.PythonProcessor (2.5.5) me mia sunˆ thsh Python grammènh se gl¸ssa sugkekrimènou tomèa tou Kale (2.5.4). Aut h sunˆ thsh Python onomˆzetai automl_orchestrate kai ousiastikˆ perigrˆfei olìklh h thn leitourgikìthta mhqanik c mˆ hshc tou Enorqhstrwt . Ekjètoume thn arqitektonik thc sunˆ thshc automl_orchestrate sthn upo •enìthta 4.2.1. To antikeÐmeno PythonProcessor (2.5.5) axiologeÐ aut thn sunˆ thsh dioqèteushc kai epist è ei èna antikeÐmenoPipeline (2.5.3).
4. MetaglwttÐ ei to antikeÐmeno Pipeline se èna Argo Work ow kai to stèlnei ston Ku e n th . Autì sum aÐnei q hsimopoi¸ntac to antikeÐmeno kale.Compiler kai thn compile_and_run mè odo tou (2.5.6).
5. Epist è ei èna antikeÐmeno AutoMLExperiment , arqikopoihmèno me to ID tou prosfˆtwc dhmiourghmènou Enorqhstrwt . Autì to antikeÐmeno epit èpei ston

q sth na parakolouje Ð thn kat ˆstash tou Enorqhstrwt kai twn Ro¸n Diamìrfwshc kai epÐshc epit èpei thn epÐbleyh twn apotelesm ˆtwn met ik c twn Ro¸n Diamìrfwshc, mìlic aut ˆ gÐnoun diajèsima .

4.1.1 To AntikeÐmeno Dataset

P okeimènou na aplopoi soume to apotÔpwma (signature) thc sun ˆ thshc run _automl() , apofasÐsame na anaparast soume to sÔnolo twn dedomènwn mhqanik c m ˆ hshc tou q sth kai ìla ta epi •mè ouc stoiqeÐa tou sac mÐa a h hmènh ontìthta . Aut h ontìthta eÐnai èna antikeÐmeno Dataset , kai pa ak ˆtw aÐnontai ta qarakthristik ˆ tou :

- ˆ name
- ˆ features
- ˆ targets
- ˆ features_test
- ˆ targets_test

4.2 O Eno qhst wt c AutoML

Se aut thn enìthta a perigr ˆyoume thn dioqèteush pou dhmiou geÐ kai elègqei olìklh o to peÐ ama AutoML. Aut h dioqèteush onom ˆzetai Enorqhstrwt c AutoML , kai ousiastik ˆ apoteleÐtai apì èxi mata ta opoÐa t èqoun k¸dika tou Kale. Pa ak ˆtw paratÐjetai mia perigraf uyhloÔ epipèdou tou mhqanismoÔ pou ulopoieÐ to k ˆ e ma :

1. get•metalearning•con gurations : Q hsimopoi¸ntac ton pu na tou auto•sklearn, autì to ma ex ˆgei èna sÔnolo apì meta •qa akth istik ˆ apì to sÔnolo dedomènwn eisìdou kai pa ˆgei mia lÐsta apì p oteinìmenec diamo ¸seic meta •m ˆ hshc .
2. run•metalearning•con gurations : Autì to ma paÐ nei tic diamorf¸seic tou - matoc 1 kai dhmiou geÐ mÐa nèa Ro Diamìrfwshc gia k ˆ e mia apì autèc .
3. monitor•kfp•runs : Autì to ma perimènei tic Roèc Diamìrfwshc pou dhmiou g hkan sto ma 2 na oloklh w oÔn .
4. get•best•con guration : Autì to ma sugkent ¸nei ta sko met ik c apì k ˆ e Ro Diamì wshc kai epilègei thn Ro RÔ mishc pou eÐqe thn kalÔte h epÐdosh .
5. run•katib•experiment : Se autì to ma , to Kale paÐ nei tic diamorf¸seic meta • m ˆ hshc pou antistoiqoÔn sto kalÔte h Ro Diamìrfwshc tou matoc 4, kai dhmiou geÐ èna peÐ ama Katib pou a ulopoieÐ diamìrfwsh upe •pa amèt wn gia to montèlo .
6. monitor•katib•experiment : Autì to ma perimènei to peÐ ama Katib tou prohgoÔ- menou matoc na oloklhrwjeÐ .

Figure 4.1: Pa ˆdeigma mÐac dioqèteushc Eno qhst wt AutoML apì thn diepa q sth tou KFP

4.2.1 H Sunˆ thsh Dioqèteushc automl_orchestrate

H sunˆ thsh automl_orchestrate eÐnai mia sunˆ thsh Python, grammènh sthn gl¸ssa sugkekrimènou tomèa tou Kale (2.5.4), h opoÐa ousiastikˆ perigrˆfei thn dom tou Enorqh-strwt AutoML pou dhmiou geÐ h sunˆ thsh run_automl . Efîson eÐnai grammènh sthn gl¸ssa sugkekrimènou tomèa tou Kale, qrhsimopoieÐ Step antikeÐmena(2.5.2) gia na ana-parast sei mata se mÐa dioqèteush . Pa akˆtw parajètoume ton k¸dika thc sunˆ thshc :

Listing 4.2: H sunˆ thsh dioqèteushc 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.dillpkl" ),
```

```
6             ml_assets _marshal _path ("task.dillpkl"          ),
7             ml_assets _marshal _path ("metric.joblib"         ),
8             ml_assets _marshal _path ("number _of _configurations.dillpkl"              ))
9
10        run _ids  = RunMetaLearningConfigurations          ()(
11            configurations       ,
12            ml_assets _marshal _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 _marshal _path ("metric.joblib"         ))
19        if hp  _tune  == "true"   :
20            katib  _experiment  _name = RunKatibExperiment     ()(
21                ml_assets _marshal _path ("tuner.dillpkl"          ),
22                best _configuration       ,
23                ml_assets _marshal _path ("metric.joblib"         ),
24                kale _dataset  _id )
25            MonitorKatibExperiment        ()(katib  _experiment  _name)
```

Pa ath ste ìti me ikèc apì tic pa amèt ouc eisìdou twn hmˆtwn eÐnai monopˆtia se seiriopoihmèna antikeÐmena pou antistoiqoÔn se apojhkeumènec pa amèt ouc eisìdou thc sunˆ thshc run_automl (4.1). 'Opwc eÐnai anerì apì ton prohgoÔmeno k¸dika , h dioqè-teush tou Enorqhstrwt AutoML thn opoÐa perigrˆfei h automl_orchestrate() , apoteleÐtai apì èxi Kale Steps:

1. GetMetaLearningCon gurations
2. RunMetaLearningCon gurations
3. MonitorKFPRuns
4. GetBestCon guration
5. RunKatibExperiment
6. MonitorKatibExperiment

Kˆe èna apì autˆ eÐnai èna kanonikì ma dioqèteushc Kale pou ulopoieÐ èna sug-kekrimèno tm ma tou Enorqhstrwt AutoML. Stic epìmenec enìthtec , a ekjèsoume thn leitourgikìthta kajenìc apì autˆ ta mata .

4.2.2 To B ma GetMetaLearningCon gurations

Se aut thn upoenìthta parousiˆzoume thn leitourgikìthta pou ulopoieÐ to ma Get• MetaLearningCon gurations .

Listing 4.3: H mè odoc do_run() tou matoc GetMetaLearningCon gurations

```
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 H Sunˆ thsh compute_con gs

H diadikasÐa upologismoÔ twn diamorf¸sewn mhqanik c mˆ hshc ulopoieÐtai ousiastikˆ apì mia sunˆ thsh tou domostoiqeÐou kale.ml.metalearning , pou onomˆzetai : compute_con gs . Pa akˆtw parajètoume ton k¸dika pou t èqei h sunˆ thsh compute_con gs :

Listing 4.4: H sunˆ thsh compute_con gs pou pa ˆgei mia lÐsta me proteinìmenec diamor¸seic dioqèteushc mhqanik c mˆ hshc

```
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 H Sunˆ thsh _validate_dataset

H sunˆ thsh _validate_dataset , ìpwc eÐnai emfanèc apì to ìnoma thc , elègqei to sÔnolo dedomènwn mhqanik c mˆ hshc pou pa èqei o q sthc . An kei sto domostoiqeÐo kale.ml.metalearning tou Kale kai ousiastikˆ qrhsimopoieÐ thn klˆsh InputValidator (sub•section 3.5.1) apì to domostoiqeÐo autosklearn.data.validation tou auto•sklearn gia na elègxei to sÔnolo dedomènwn eisìdou . AkoloÔ wc , parousiˆzoume ton k¸dika thc sunˆ thshc _validate_dataset .

Listing 4.5: H sunˆ thsh _validate_dataset tou domostoiqeÐou kale.ml.metalearning

```
def _validate _dataset (dataset : Dataset ,
                        task : str = "classification" ) •> (
                            Tuple [Dataset , List [str ]]):
    """Validate and process the input features and targets.

    Use AutoSKLearn ``InputValidator`` to check if the input dataset
    is valid (e.g: the number of samples matches the number of targets).
    Also, auto-sklearn does some "polishing" transformations to the dataset.

    During the validation, ``InputValidator`` also determines the feature
    type for all the input features. A feature type can either be "numerical"
    or "categorical".

    Args:
        dataset (Dataset): A Dataset class object that contains the input
            dataset for the ML task.
        task (str): The type of the ML task (classification | regression).

    Returns:
        Dataset, List(str): The validated dataset and a list of feature types
            for all features (either "numerical" or "categorical").
    """
    is _classification = (task == "classification" )
    input _validator = InputValidator ()
    x, y = input _validator .validate (X=dataset .features , y=dataset .targets ,
                                       is _classification =is _classification )
    x_test , y_test = input _validator .validate (
        X=dataset .features _test , y=dataset .targets _test ,
        is _classification =is _classification )

    validated _dataset = copy .deepcopy (dataset )
    validated _dataset .features = x
    validated _dataset .targets = y
    validated _dataset .features _test = x_test
    validated _dataset .targets _test = y_test
    return validated _dataset , input _validator .feature _types
```

4.2.2.3 H Sunˆ thsh nd_metadata_dir

H sunˆ thsh nd_metadata_dir tou domostoiqeÐou kale.ml.utils Ðskei to monopˆti ston katˆlogo meta •dedomènwn pou antistoiqeÐ se mia dosmènh ergasÐa mhqanik c mˆ hshc . Pa akˆtw aÐnetai o k¸dikac thc sunˆ thshc :

Listing 4.6: H sunˆ thsh nd_metadata_dir pou Ðskei to swstì monopˆti ston katˆlogo meta•dedomènwn gia mÐa dosmènh e gasÐa mhqanik c mˆ hshc

```
def find _metadata _dir (task _type : int , metric : Callable , is _sparse : int ):
```

```
    """Find the directory where auto-sklearn stores its meta-knowledge.

    Note:
        The directory structure follows the 'Algorithm Selection Library'
        (ASLib) format. See https://www.automl.org/automated-algorithm-design/algorithm-
      selection/aslib/  # noqa: 501

    Args:
        task_type (int): The type of the ML task.
        metric (callable): The metric that is used for the ML task.
        is_sparse (int): Whether the dataset is sparse or not.

    Returns:
        str: path to the auto-sklearn metadata directory.

    Raises:
        RuntimeError: When the auto-sklearn metadata directory cannot be found.
    """
    log.info("Finding metadata dir...")
    metalearning_directory = os.path.dirname(autosklearn.metalearning.__file__)
    # The auto-sklearn metadata directory doesn't provide metadata for
    # multi-label classification. auto-sklearn reverts to using binary
    # classification as well, so we copy this behaviour.
    if task_type == constants.MULTILABEL_CLASSIFICATION:
        meta_task = constants.BINARY_CLASSIFICATION
    else:
        meta_task = task_type
    metalearning_files_dir = "%s_%s_%s" % (
        metric, constants.TASK_TYPES_TO_STRING[meta_task],
        "sparse" if is_sparse else "dense")
    metadata_directory = os.path.join(
        metalearning_directory, "files", metalearning_files_dir)
    if not os.path.exists(metadata_directory):
        raise RuntimeError("Metadata directory %s does not exist."
                           % metadata_directory)
    log.info("Metadata directory: %s", metadata_directory)
    return metadata_directory
```

4.2.2.4 H Mè odoc _submit_con guration_artifact

P okeimènou na diathr soume mia genealogÐa olìklh ou tou peirˆmatoc AutoML, anagkˆzoume ta mata dioqeteÔsewn na dhmiourgoÔn kai na upobˆlloun Artifacts sthn ˆsh dedomènwn MLMD. H mè odoc _submit_con guration_artifact dhmiou geÐ èna AutoML• Con guration Artifact (subsubsection 4.4.1.2) pou antistoiqeÐ se mia dosmènh diamìrfwsh dioqèteushc .

Listing 4.7: H mè odoc _submit_con guration_artifact tou GetMetaLearningCon gurations pou dhmiou geÐ kai upobˆllei èna AutoMLCon guration Artifact gia mia dosmènh diamìr- wsh dioqèteushc

```
def _submit _configuration _artifact (self , configuration , idx ):
    from kale .ml import utils
    from kale .common.artifacts import AutoMLConfiguration

    mlmd = mlmdutils .get _mlmd_instance ()
    config _summary = utils .get _configuration _summary(configuration )
    config = AutoMLConfiguration (
        config _summary=config _summary,
        run _id =kfputils .format _kfp _run _id _uri (mlmd.run _uuid ),
        estimator _name="Configuration %s: %s"
                        % (idx + 1, config _summary["name" ]))
    config .assign _list _index (idx + 1)
    config _artifact = config .submit _artifact ()
    mlmd.link _artifact _as_output (config _artifact .id )
```

Gia na pa ˆxei thn pˆ amet o con g_summary , to ma autì qrhsimopoieÐ thn sunˆ thsh get_con guration_summary tou domostoiqeÐou ml.utils :

Listing 4.8: H sunˆ thsh get_con guration_summary pou dhmiou geÐ èna lexikì •pe Ðlhyh miac dosmènhc diamì wshc dioqèteushc

```
def get _configuration _summary(configuration : Configuration ) •> Dict [str , Any]:
"""Return an opinionated summary of the input configuration.

The output of this function can be used to pretty-print a configuration,
with just the right information, or to upload the configuration to the
artifact store.

Args:
    configuration: A suggested configuration extracted by auto-sklearn.

Returns:
    dict: A dictionary that describes the learner (classifier, or
        regressor) of the configuration, with the following fields:

        * ``name``: The name of the model
        * ``parameters``: A dictionary of hyperparameters.

Raises:
    ValueError: If cannot find a supported learner type. Supported
        learner types are ``classifier: __choice __`` and
        ``regressor: __choice __``.
"""
if configuration .get ("classifier: __choice __"):
    name = _get _classifier _name(configuration )
    params = _get _params (configuration , "classifier" )
```

```
elif configuration .get ("regressor: __choice __" ):
    name = _get _regressor _name(configuration )
    params = _get _params (configuration , "regressor" )
else :
    raise ValueError ("Could not find a model in the input configuration." )

return {"name" : name, "parameters" : params }
```

4.2.3 To B ma RunMetaLearningCon gurations

Se aut n thn upoenìthta a parousiˆsoume kai a exhg soume ton mhqanismì pou ulopoieÐ to ma RunMetaLearningCon gurations . Sunoptikˆ , autì to ma lambˆnei wc eÐsodo thn lÐsta twn diamorf¸sewn dioqèteushc pou dhmioÔ ghse to prohgoÔmeno ma , kai gia kˆ e mia touc , dhmiou geÐ mÐa nèa Ro Diamìrfwshc pou ulopoieÐ thn dioqèteush mhqanik c mˆ hshc pou h antÐstoiqh diamìrfwsh dioqèteushc perigrˆfei . Ac doÔme ton k¸dika pou ekteleÐtai mèsa sto ma autì , xekin¸ntac apì ton o ismì thc klˆshc tou kai thn mè odo do_run :

Listing 4.9: O orismìc klˆshc kai h mè odoc do_run() tou matoc RunMetaLearningCon• gurations

```
class RunMetaLearningConfigurations (Step ):
    """Run MetaLearning suggestions as KFP pipelines.

    Ins:
        configurations (List[Configuration])
        max_parallel _configurations (int)
        kale _dataset _id (int)

    Outs:
        run _ids (List[str])
    """
    name = "run-metalearning-configurations"
    outs = Param.odict ([("run _ids" )], step _name=name)
    actions = ["RunKFPPipelines" ]

def do _run (self , configurations , max_parallel _configurations ,
             kale _dataset _id ):
    """Implementation of RunMetaLearningConfigurations."""
    from time import sleep
    from kale .common import mlmdutils
    from kale .ml.utils import ML _ASSETS_DIR
    from kale .common.artifacts import AutoMLConfiguration

    self .vars ["run _ids" ] = []
    marshal .set _data _dir (ML_ASSETS_DIR)

```

```
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 H Mè odoc _run_pipeline

'Otan o a i mìc twn ekteloÔmenwn Ro¸n Diamì wshc eÐnai mik ìte oc apì thn tim thc pa amèt ou max_parallel_con gurations (pa ˆmet oc eisìdou thc sunˆ thshc run_• automl), to ma RunMetaLearningCon gurations epilègei mia diamì wsh dioqèteushc kai dhmiou geÐ mÐa nèa Ro RÔ mishc pou na thn ulopoieÐ . H dhmiou gÐa thc Ro c RÔ mishc

ulopoieÐtai apì thn mè odo _run_pipeline thc klˆshc RunMetaLearningCon gurations .

Listing 4.10: 'H mè odoc _run_pipeline thc klˆshc RunMetaLearningCon gurations

```
def _run _pipeline (self , index , params : Dict = {}):
        from kale .types import Param
        from kale import PipelineConfig , Compiler
        from kale .processors import PythonProcessor
        from kale .ml.pipelines import sklearn _train _predict

        volumes = rokutils .interactive _snapshot _and_get _volumes ()
        pipeline _config = PipelineConfig (
            pipeline _name="sklearn-configuration-%d" % index ,
            experiment _name=kfputils .get _experiment _from _run _id (
                kfputils .detect _run _uuid ()).name,
            marshal _path =self .marshal _path ,
            volumes =volumes )
        pipeline _params = {k: Param(type (v).__name__, v)
                            for k , v in params .items ()}

        log .newline ()
        log .info ("Creating pipeline for configuration n. %d" , index )
        processor = PythonProcessor (
            sklearn _train _predict .sklearn _train _predict , pipeline _config )
        processor .pipeline .default _pipeline _parameters .update (pipeline _params )
        pipeline = processor .run ()
        log .info ("Running pipeline for configuration n. %d" , index )
        run = Compiler (pipeline ).compile _and_run ()
        log .info ("Successfully started run %s" % run .id )
        log .newline ()
        return run .id
```

'Opwc perigrˆyame parapˆnw , h mè odoc _run_pipeline ousiastikˆ dhmiou geÐ pragmatikèc dioqeteÔseic Kube ow, oi opoÐec onomˆzontai Roèc RÔ mishc. Gia ton skopì autì h mè odoc ousiastikˆ qrhsimopoieÐ to back•end tou Kale (section 2.5).

4.2.4 To B ma MonitorKFPRuns

Se aut thn upoenìthta a ekjèsoume ton mhqanismì pou ulopoieÐ to ma MonitorKF•PRuns. Sunoptikˆ , autì to ma parakoloujeÐ thn katˆstash ìlwn twn Ro¸n Diamìrfwshc kai mìlic oloklh w oÔn ìlec touc , epituqhmèna ìqi , to ma oloklh¸nei thn ektèlesh tou. Ac parousiˆsoume ton k¸dika pou t èqei mèsa se autì to ma , kai pio sugkekrimèna ton o ismì klˆshc tou kai thn mè odo do_run :

Listing 4.11: O o ismìc klˆshc kai h mè odoc do_run() tou matoc MonitorKFPRuns

```
class MonitorKFPRuns (Step ):
    """Wait for KFP pipelines to complete.

    Ins:
        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
```

'Opwc mpo oÔme na doÔme ston pa apˆnw k¸dika , to apotÔpwma thc me ìdou do_run tou matoc eÐnai a ketˆ apl . Lam ˆnei mÐa lÐsta apì IDs Ro¸n Diamì wshc wc eÐsodo , kai thn epist è ei mìlic telei¸sei thn ektèlesh tou , ¸ste na thn lˆ ei to epìmeno ma . H lÐsta me ta IDs twn Ro¸n Diamì wshc a q hsimopoih eÐ gia na e wth eÐ o KFP server gia thn katˆstash twn Ro¸n Diamì wshc .

4.2.5 To B ma GetBestCon guration

Se aut thn upoenìthta a parousiˆsoume kai a exhg soume ton mhqanismì pou ulopoieÐ to ma GetBestCon guration . Sunoptikˆ , autì to ma paÐ nei wc eÐsodo thn lÐsta me ta IDs twn Ro¸n Diamìrfwshc , sullègei ta sko met ik c twn epituqhmènwn Ro¸n Diamìrfwshc , Ðskei thn Ro Diamìrfwshc me to kalÔte o sko kai epist è ei to antÐs-toiqo antikeÐmeno diamìrfwshc mhqanik c mˆ hshc .

Listing 4.12: O o ismìc klˆshc kai h mè odoc do_run() tou matoc GetBestCon guration

```
class GetBestConfiguration (Step):
    """Get the best-performing MetaLearning configuration.

    Ins:
        run_ids
        configurations
        metric

    Outs:
        best_configuration
    """
    name = "get-best-configuration"
    outs = Param.odict(["best_configuration"], step_name=name)

    def do_run(self, run_ids, configurations, metric):
        """Implementation of GetBestConfiguration."""
        metrics = dict()
        for run_id in run_ids:
            log.info("Collecting metrics for run: %s", run_id)
            metrics[run_id] = kfputils.get_kfp_run_metrics(run_id)
        final_metrics = metrics.copy()

        log.newline()
        log.info("Collected metrics:\n")
        for run_id, _metrics in metrics.items():
            log.info("  Run %s:", run_id)
            if not _metrics.values():
                log.info("    No metrics found.")
                del final_metrics[run_id]
            for name, value in _metrics.items():
                log.info("    %s: %s", name, value)
        log.info("Using metric '%s' as target metric.", metric.name)

        # Get best metric, excluding empty metrics dictionaries
        opt = max if metric._sign == 1 else min  # see arrikto/dev#1128
        best_run_uid, best_metric = opt(final_metrics.items(),
                                        key=lambda x: x[1][metric.name])
        log.info("Best run id: %s", best_run_uid)

        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 To B ma RunKatibExperiment

Se aut n thn upoenìthta , a pa ousiˆsoume ton mhqanismì pou ulopoieÐ to ma RunKatibExperiment . To ma autì paÐ nei thn diamì wsh dioqèteushc me to megalÔte o sko met ik c apì to p ohgoÔmeno ma , kai dhmiou geÐ èna peÐ ama (2.4.4) ¸ste na elti¸sei pe aitè w thn dioqèteush aut .

Autì kai to epìmeno ma (MonitorKatibExperiment) ekteloÔntai mìno sthn pe Ðptwsh pou o q sthc èqei orÐsei èna antikeÐmeno tuner (katib.ExperimentSpec object) wc pa ˆmet o eisìdou sthn sunˆ thsh run_automl (section 4.1). An den èqei pe asteÐ kˆpoia pa ˆmet oc eisìdou tuner sthn sunˆ thsh run_automl , tìte to GetBestCon guration (sub• section 4.2.5) eÐnai to teleutaÐo ma tou Enorqhstrwt AutoML. Pa akˆtw parousiˆzoume ton o ismì klˆshc kai tic kÔ iec mejìdouc tou matoc :

Listing 4.13: O o ismìc klˆshc kai oi kÔ iec mè odoi tou matoc 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 To B ma MonitorKatibExperiment

Autì eÐnai to teleutaÐo ma tou Enorqhstrwt AutoML kai o skopìc tou eÐnai na parakolouje Ð to peÐ ama Katib pou dhmioÔ ghse to prohgoÔmeno ma .

Listing 4.14: The class de nition and do_run method of the MonitorKatibExperiment step

```
class MonitorKatibExperiment (Step ):
    """Wait for a Katib experiment to complete.

     Ins:
         katib _experiment _name (str): Name of the Katib experiment to monitor.
     """
    name = "monitor-katib-experiment"

    def do _run (self , katib _experiment _name: str ):
        """Implementation of MonitorKatibExperiment."""
        from kale .common import katibutils

        katibutils .wait _for _hptuning _experiment (katib _experiment _name)
```

'Opwc eÐnai anerì parap nw , h mè odoc do_run tou MonitorKatibExperiment ousi- astik perimènei to peÐ ama Katib na telei¸sei thn ektèlesh tou . To k nei autì kal¸n- tac thn ohjhtik sun thsh wait_for_hptuning_experiment apì to domostoiqeÐo com• mon.katibutils tou Kale. Pa ak tw aÐnetai o k¸dikac thc sun thshc :

Listing 4.15: H oh htik sun thsh wait_for_hptuning_experiment tou domostoiqeÐou kale.common.katibutils

```
def wait _for _hptuning _experiment (experiment _name: str ):
    """Wait for an HP Tuning experiment to succeed.

     Args:
         experiment _name (str): Name of the HP Tuning experiment.

     Returns:
         tuple(str, str, str): Status, Condition reason, Condition message.
     """
    def sleep _with _progress (total , interval , msg):
        for i in range (0, total // interval ):
            log .info ("%s %d..." , msg, total • i * interval )
            time .sleep (interval )

    while True :
        log .newline (2)
        log .info ("Watching for HP Tuning experiment: '%s'" ,
                   experiment _name)
        sleep _with _progress (30, 5, "Checking status in" )

        experiment = get _experiment (experiment _name, podutils .get _namespace ())
        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 Oi Roèc Diamì wshc

Se aut thn enìthta , a perigrˆyoume thn arqitektonik twn Ro¸n Diamìrfwshc . P ìkeitai gia dioqeteÔseic tic opoÐec dhmiou geÐ o Enorqhstrwt c AutoML katˆ thn diˆ keia tou - matoc RunMetaLearningCon gurations , ètsi ¸ste na ekpaideÔsei montèla pa ˆllhla . Kˆ e Ro Diamìrfwshc se èna peÐ ama AutoML ulopoieÐ mÐa apì tic diamorf¸seic dioqèteushc pou pa ˆgei to auto•sklearn.

Autèc oi dioqeteÔseic ousiastikˆ apoteloÔntai apì t Ða mata , kai kˆ e ma anti- stoiqeÐ se èna sugkekrimèno tm ma miac o c ergasÐac mhqanik c mˆ hshc . Pa akˆtw paratÐjetai mia perÐlhyh thc leitourgÐac tou kˆ e matoc :

1. run•sklearn•transformer : Autì to ma ulopoieÐ ton p o •epexe gast pou peri- grˆfei mia diamìrfwsh dioqèteushc . O skopìc tou eÐnai na epexergasteÐ to sÔnolo dedomènwn eisìdou (tìso to sÔnolo ekpaÐdeushc ìso kai to sÔnolo exètashc) kai na to è ei se mia katˆstash pou a epit èyei sto montèlo na ekpaideuteÐ kai na exetasjeÐ h apìdosh tou pˆnw se autì to sÔnolo dedomènwn .

2. train•sklearn•estimator : Autì to ma ousiastikˆ ulopoieÐ thn arqitektonik tou montèlou pou perigrˆfei mia diamìrfwsh dioqèteushc . O skopìc tou eÐnai na pa ˆxei èna ekpaideumèno montèlo qrhsimopoi¸ntac to epexergasmèno sÔnolo dedomènwn ek- paÐdeushc pou pa ˆgei to prohgoÔmeno ma .

3. infer•sklearn•predictor : Autì to ma qrhsimopoieÐ to epexergasmèno sÔnolo de- domènwn ekpaÐdeushc tou matoc run•sklearn•transformer gia na elègxei thn apì- dosh tou montèlou pou pa ˆq hke sto ma train•sklearn•estimator qrhsimopoi¸n- tac thn sunˆ thsh met ik c pou o q sthc pa eÐqe . 'Otan ta sko met ik c apì autì to ma èqoun telikˆ paraqjeÐ , olìklh h h Ro Diamìrfwshc oloklh ¸nei thn ek- tèlesh thc .

Pa akˆtw parousiˆzoume èna pa ˆdeigma miac Ro c Diamìrfwshc pou èqei oloklhr¸- sei thn leitourgÐa thc , ìpwc aÐnetai sthn diepa q sth tou KFP.

Figure 4.2: Pa ˆdeigma oloklh wmènhc Ro c Diamì wshc sthn diepa q sth tou KFP

'Opwc ana è ame se sthn upoenìthta 4.2.3, h sunˆ thsh dioqèteushc pou qrhsimopoieÐtai san makèta gia tic Roèc Diamìrfwshc eÐnai h sklearn_train_predict() apì to domostoiqeÐo kale.ml.pipelines . Sthn epìmenh upoenìthta parajètoume kai analÔoume aut n thn sunˆ thsh dioqèteushc .

4.3.1 H Sunˆ thsh Dioqèteushc sklearn_train_predict

H sunˆ thsh sklearn_train_predict eÐnai mia sunˆ thsh dioqèteushc Python, grammènh sthn gl¸ssa sugkekrimènou tomèa tou Kale (2.5.4), h opoÐa ousiastikˆ perigrˆfei thn dom thc Ro c Diamìrfwshc pou to dhmiou geÐ to ma RunMetaLearningCon gurations (4.2.3). Efìson eÐnai grammènh se gl¸ssa eidikoÔ skopoÔ tou Kale, qrhsimopoieÐ antikeÐmenaStep (2.5.2) tou Kale gia na anaparast sei mata sthn dioqèteush . Pa akˆtw aÐnetai o k¸dikac thc sunˆ thshc :

Listing 4.16: H sunˆ thsh dioqèteushc 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 )
```

'Opwc aÐnetai apì ton p ohgoÔmeno k¸dika , h dioqèteush miac Ro c Diamì wshc pou pe ig ˆ ei h sunˆ thsh sklearn_train_predict , apoteleÐtai apì t Ða mata :

1. RunSKLearnTransformer
2. TrainSKLearnEstimator
3. InferSKLearnPredictor

Kˆ e èna apì autˆ eÐnai èna ma Kale pou ulopoieÐ èna sugkekrimèno tm ma thc dioqèteushc miac Ro c Diamìrfwshc . Stic akìloujec upoenìthtec , a ekjèsoume thn lei- tourgikìthta kajenìc apì autˆ ta mata .

4.3.2 To B ma RunSKLearnTransformer

Se aut thn upoenìthta , a ekjèsoume kai a exhg soume ton mhqanismì pou ulopoieÐ to ma RunSKLearnTransformer . Ja perigrˆ youme pwc autì to ma diabˆ zei wc eÐsodo mia diamìrfwsh dioqèteushc kai ulopoieÐ to ma tou p o •epexe gast pou perigrˆ fei h diamìrfwsh aut . Pa akˆ tw , ekjètoume ton o ismì klˆ shc kai tic kÔ iec mejìdouc tou matoc :

Listing 4.17: O o ismìc klˆ shc kai oi kÔ iec mè odoi tou matoc 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.marshal.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" ])
```

'Opwc kai sthn pe Ðptwsh twn hmˆtwn tou Eno qhst wt AutoML, h mè odoc do_run() eÐnai o kÔ ioc k¸dikac pou t èqei sto eswte ikì tou matoc . Epomènwc, ac ek èsoume mia pe Ðlhyh thc logik c pou ulopoieÐ aut h mè odoc . H mè odoc do_run :

1. O Ð ei ta Artifacts Dataset kai AutoMLCon guration wc eisìdouc autoÔ tou matoc . O o ismìc autìc ulopoieÐtai me q sh thc me ìdou _link_input_artifacts tou matoc .
2. Dia ˆ ei thn diamìrfwsh dioqèteushc pou dèqetai wc eÐsodo . An h t èqousa Ro Diamìrfwshc eÐnai mè oc enìc peirˆmatoc Katib, (dhmiourghmènh apì to ma RunKatibExperiment tou Enorqhstrwt AutoML) tìte anane¸noume thn diamìrfwsh dioqèteushc , pou èqei dhmiourghjeÐ apì to auto•sklearn, me tic timèc pou proteÐnei to Katib. Sthn pe Ðptwsh pou h t èqousa Ro Diamìrfwshc èqei dhmiourghjeÐ apì to ma RunMetaLearningCon gurations , k atˆme thn diamìrfwsh dioqèteushc wc èqei. Ulopoi same aut n thn apìfash sthn sunˆ thsh patch_con guration tou do-mostoiqeÐoukale.ml.utils :

   Listing 4.18: H sunˆ thsh patch_con guration tou domostoiqeÐou kale.ml.utils

```
def patch_configuration(configuration: Configuration,
                        overrides: Dict[str, Any],
                        coerce=False):
"""Patch a configuration.

Args:
    configuration: A suggested configuration extracted by auto-sklearn.
    overrides (dict): Override configuration values by providing them with
        keys matching the last token in the original Configuration keys.
        Provide an empty dictionary to make this function a no-op.
    coerce (bool): Convert the override value to the destination type.

Returns:
    Configuration: Patched configuration.
"""
log.info("Patching base configuration...")
log.info("Base %s", configuration)

if not overrides:
    log.info("No input parameters found to patch the base"
             " configuration.")
    return configuration

log.info("Using the following configs to patch the base"
         " configuration: %s", overrides)
patched_config = copy.deepcopy(configuration)
for name, override_value in overrides.items():
    for key, conf_value in patched_config.get_dictionary().items():
        if key.split(":")[-1] == name:
            if coerce:
                override_value = type(conf_value)(override_value)
            patched_config[key] = override_value
log.info("Patched %s", patched_config)
return patched_config

```

3. Metatèpei thn diamìrfwsh dioqèteushc se mÐa taxinomhmènh lÐsta me mata. Gia ton skopì autì ulopoi same thn sunˆ thsh get_sklearn_steps_list tou domostoiqeÐoukale.ml.utils :

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

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

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

4. A ai eÐ to teleutaÐo ma pou antistoiqeÐ sto montèlo kai k atˆei thn up-ìloiph lÐsta .
5. Dhmiou geÐ èna antikeÐmeno sklearn.pipeline Pipeline object me thn lÐsta pou t¸ a pe ièqei mìno mata p o •epexe gasÐac .
6. Epexe gˆ etai to sÔnolo dedomènwn kˆnontac q sh twn me ìdwn transform kai t_transform (2.1.1) tou antikeimènou Pipeline tou sklearn.
7. Dhmiou geÐ èna Transformer Artifact (4.4.1.3) kai to o Ð ei wc èxodo tou matoc q hsimopoi¸ntac thn sunˆ thsh _link_output_artifacts .
8. Epist è ei to epexe gasmèno sÔnolo dedomènwn gia to epìmeno ma thc Ro c Diamì wshc .

4.3.3 To B ma TrainSKLearnEstimator

Se aut n thn upoenìthta , a ekjèsoume kai a exhg soume ton mhqanismì pou ulopoieÐ to ma TrainSKLearnEstimator . Genikˆ, h mè odoc do_run tou matoc akolouje Ð thn Ðdia logik me thn do_run tou prohgoÔmenou matoc . Ja perigrˆyoume pwc to ma Train• SKLearnEstimator diabˆzei thn diamìrfwsh dioqèteushc pou dèqetai wc eÐsodo , kataskeuˆ ei to montèlo pou perigrˆfei h diamìrfwsh kai Ôste a ekpaideÔei to montèlo pˆnw sto epex-ergasmèno sÔnolo dedomènwn ekpaÐdeushc pou dhmioÔ ghse to prohgoÔmeno ma . Ac doÔme ton o ismì klˆshc kai ton k¸dika thc mejìdou do_run :

Listing 4.20: O o ismìc klˆshc kai oi asikèc mè odoi tou matoc TrainSKLearnEstimator

```
class TrainSKLearnEstimator (PatchMaskedInputsMixin , Step ):
    """Train a SKLearn estimator from an AutoML configuration.

    Ins:
        configuration:
        x_processed:
        dataset:
        kale _dataset _id _local:

    Outs:
        model: Trained model
        kale _model _id: MLMD artifact ID of the trained model

    MLMD Inputs:
        kale.Configuration

    MLMD Outputs:
        kale.Model
    """
```

```
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
```

4.3.4 To B ma InferSKLearnPredictor

Se aut thn upo •enìthta a ekjèsoume kai a perigrˆyoume ton mhqanismì pou ulopoieÐ to ma InferSKLearnPredictor . Sunoptikˆ , h mè odoc do_run tou matoc autoÔ diabˆzei to ekpaideumèno montèlo pou dèqetai wc eÐsodo apì to prohgoÔmeno ma , kai Ôste a elègqei to montèlo pˆnw sto epexergasmèno sÔnolo dedomènwn exètashc pou dhmioÔ ghse to ma RunSKLearnTransformer . Ac deÐxoume ton o ismì klˆshc kai tic kÔ iec mejìdouc tou matoc :

Listing 4.21: O o ismìc klˆshc kai oi kÔ iec mè odoi tou matoc InferSKLearnPredictor

```
1  class InferSKLearnPredictor        (Step ):
```

```
    """Run predictions on a trained model.

    Ins:
        model
        x_test _processed
        metric
        dataset
        task
        kale _model _id
        kale _dataset _id _local

    MLMD Inputs:
        kale.Model
        kale.Dataset

    MLMD Outputs:
        kale.TensorboardLogs
    """
    name = "infer-sklearn-predictor"
    has_metrics = True

    def _link _input _artifacts (self , kale _dataset _id , kale _model _id ):
        mlmd = mlmdutils .get _mlmd_instance ()
        mlmd.link _artifact _as_input (kale _dataset _id )
        mlmd.link _artifact _as_input (kale _model _id )

    def _update _model _artifact (self , kale _model _id , metric _name, metric _value ):
        mlmd = mlmdutils .get _mlmd_instance ()
        model = mlmdutils .get _artifact _by_id (kale _model _id )
        model _metrics = json .loads (model .properties ["metrics" ].string _value )
        model _metrics .update ({metric _name: metric _value })

        model _name = model .properties ["name" ].string _value
        class _name = model _name.split ("/" )[-1]
        new_name = "%s/%s" % (mlmd.name or "" , class _name)

        mlmdutils .patch _artifact _properties (kale _model _id ,
                                                {"name" : new_name,
                                                 "metrics" : model _metrics })

    def _produce _report (self , task , dataset , model ):
        from kale .ml import visualizations
        from tensorboardX import SummaryWriter

        log .info ('Logging report to Tensorboard...' )

        viz = visualizations .SklearnVisualizer (task =task , dataset =dataset ,
                                                 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 H GenealogÐa MLMD kai h Pa akoloÔ hsh Katˆstashc twn Pei amˆtwn AutoML tou Kale

Katˆ thn diˆ keia thc anˆptuxhc thc diergasÐac AutoML tou Kale, èlame na dhmiourg - soume mia apì •ˆk h •se•ˆk h genealogÐa twn pei amˆtwn AutoML. O telikìc stìqoc tan na epitrèyoume ston q sth , pou kaleÐ thn sunˆ thsh run_automl , na lèpei mia olìklh h perÐlhyh tou dhmiourghmènou peirˆmatoc AutoML. H perÐlhyh aut perikleÐei :

- ˆ thn katˆstash tou Eno qhst wt AutoML
- ˆ thn katˆstash twn Ro¸n Diamì wshc
- ˆ thn katˆstash tou peirˆmatoc Katib sthn pe Ðptwsh pou parèqetai èna antikeÐ- meno eisìdou tuner sthn sunˆ thsh run_automl

EÐqame wc stìqo na mpo oÔme na èqoume epopteÐa ìlwn twn parapˆnw , apì mia kl sh sunˆ thshc run_automl pou thc parèqetai to ID thc dioqèteushc tou Enorqhstrwt Au• toML.

Dhmiou g¸ntac mia genealogÐa sto MLMD, kata è ame na parakolouj soume thn katˆstash olìklh ou tou peirˆmatoc AutoML, xekin¸ntac apì to ID tou Enorqhstrwt AutoML.

4.4.1 Artifacts tou Kale

Se aut n thn enìthta a exetˆsoume touc diaforetikoÔc tÔpouc apì Artifacts pou dhmiou geÐ to Kale ¸ste na diathr sei mia genealogÐa MLMD gia ìlh thn diˆ keia enìc peirˆmatoc AutoML.

Gia na mpo oÔme na upobˆlloume kai na anane¸noume diaforetik¸n eid¸n MLMD Ar• tifacts, dhmiou g same to domostoiqeÐo kale.common.artifacts to opoÐo perièqei orismoÔc klˆsewn gia ìla ta Artifacts pou a qreiastoÔme . Ja ekjèsoume autèc tic klˆseic stic akìloujec upoenìthtec .

4.4.1.1 H Basik Klˆsh MLMDArtifact

Aut eÐnai h asik klˆsh gia antikeÐmena pou upo ˆllontai sthn ˆsh tou MLMD. 'Allec klˆseic pou antistoiqoÔn se Artifacts tou MLMD kai twn opoÐwn ta antikeÐmena p èpei na upo lh oÔn sto MLMD, p èpei na klh onomoÔn apì aut thn klˆsh kai p èpei na :

1. èqoun gnw Ðsmata artifact_type_name kai artifact_property_types
2. ulopoioÔn tic akìlou ec idiìthtec : artifact_properties kai artifact_custom_• properties
3. èqoun èna gn¸ isma artifact_uri , pou a èqei tim katˆ thn diˆ keia tou qrì- nou ektèleshc
4. ulopoioÔn thn mè odo submit_artifact

Listing 4.22: H asik klˆsh MLMDArtifact pou q hsimopoieÐtai gia thn upo ol Artifacts sto MLMD

```
class MLMDArtifact (abc.ABC):
    """A base class for objects submittable in MLMD,

    Other classes that correspond to MLMD Artifacts and that expect their instances to be
    submitted in MLMD should subclass this class.

    Example:

        >>> class MyClass(MLMDArtifact):
        ...
        ...     artifact_type_name = "kale.MyClass"
        ...     artifact_property_types = {"prop1": metadata_store_pb2.STRING,
        ...                                "prop2": metadata_store_pb2.INT}
        ...
        ...     @property
        ...     def artifact_properties(self):
        ...         prop1 = metadata_store_pb2.Value(
        ...             string_value=self.get_prop1())
        ...         prop2 = metadata_store_pb2.Value(
        ...             int_value=self.compute_prop2())
        ...         return {"prop1": prop1, "prop2": prop2}
        ...
        ....    @property
        ...     def artifact_custom_properties(self):
        ...         custom_prop = metadata_store_pb2.Value(
        ...             string_value="custom_value")
        ...         return {"custom_prop": custom_prop}
        ...
        >>> instance = MyClass()
        >>> instance.artifact_uri = "gs://bucket/path/to/blob"
        >>> artifact = instance.submit_artifact()

    For more information on available property types, see
    https://github.com/google/ml-metadata/blob/v0.29.0/ml_metadata/proto/metadata_store.
    proto#L74-L81

    Attributes:
        artifact_type_name (str): The name of the ArtifactType
        artifact_property_types (dict): The mapping of property names and their
            MLMD types
        artifact_uri (str): The URI of the Artifact
    """

    artifact_type_name: str = "kale.Artifact"
    artifact_property_types : Dict = None
    artifact_uri : str = None

    @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 AutoMLCon guration

'Enac apì touc stìqouc mac , sqetikˆ me thn genealogÐa MLMD tou peirˆmatoc AutoML, tan na èqoume thn dunatìthta na anaktoÔme plhroforÐec sqetikèc me tic proteinìmenec di-amorf¸seic dioqèteushc pou pa ˆgei to auto•sklearn. Pio analutikˆ , èlame na mpo oÔme na lambˆnoume aut n thn plhroforÐa qrhsimopoi¸ntac aplˆ to ID tou Enorqhstrwt Au• toML.

Gia na to ulopoi soume autì , apofasÐsame ìti ìtan to ma get•metalearning•con guration tou Enorqhstrwt ekteleÐtai , p èpei na dhmiou geÐ AutoMLCon guration Artifacts (sub• section 4.2.2). Autì to ma epÐshc o Ð ei autˆ ta Artifacts wc exìdouc tou . Autì sumbaÐnei upobˆllontac MLMD Events (2.6) pou orÐzoun ìti ta AutoMLCon guration Artifacts eÐnai èxodoi tou Execution pou antistoiqeÐ sto ma get•metalearning•con guration .

Autˆ ta AutoMLCon guration Artifacts èqoun mia kÔ ia idiìthta pou onomˆzetai model_• data . P in h mè odoc do_run tou matoc GetMetaLearningCon gurations upobˆllei èna AutoMLCon guration Artifact sto MLMD, " gemÐ ei aut n thn idiìthta (4.2.2.4) me èna lexikì pou perièqei :

1. To ìnoma thc arqitektonik c tou montèlou thc antÐstoiqhc diamìrfwshc dio-qèteushc .
2. Ta onìmata kai tic timèc twn pa amèt wn tou montèlou .

O akìloujoc k¸dikac perièqei ton o ismì thc klˆshc , ta gnwrÐsmata kai tic mejìdouc tou AutoMLCon guration :

Listing 4.23: H klˆsh AutoMLCon guration pou qrhsimopoieÐtai gia dhmiou gÐa kai up-obol Artifacts pou antistoiqoÔn se diamorf¸seic dioqèteushc

```
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 'Alla Artifacts

To Kale epÐshc dhmiou geÐ kai anane¸nei èna pl oc apì ˆlla MLMD Artifacts, kˆe èna apì ta opoÐa antistoiqeÐ se mia sugkek imènh "ontìthta " thc die gasÐac AutoML tou Kale. E ìson den a estiˆsoume ku Ðwc se autˆ se aut n thn diplwmatik e gasÐa , a ek èsoume sÔntoma autˆ ta Artifacts kai ton skopì touc , sthn akìlou h lÐsta :

- ˆ Dataset : 'Ena Artifact gia pa akoloÔ hsh thc katˆstashc tou sunìlou dedomènwn mhqanik c mˆ hshc .
- ˆ Model : 'Ena Artifact gia apo keush tou ekpaideumènou montèlou pou paˆgetai katˆ to ma train•sklearn•estimator (4.3.3) miac Ro c Diamì wshc .
- ˆ Transformer : 'Ena Artifact gia apo keush tou pre•processor pou paˆgetai katˆ to ma run•sklearn•transformer (4.3.2) miac Ro c Diamì wshc .

4.4.2 To AntikeÐmeno AutoMLExperiment

Autì to antikeÐmeno eÐnai to kÔ io e galeÐo gia thn pa akoloÔ hsh thc katˆstashc kai thc exèlixhc tou pei ˆmatoc AutoML. Dedomènou touID tou Eno qhst wt AutoML, èna antikeÐmenoAutoMLExperiment è nei kai apeikonÐ ei ìlec tic plh o o Ðec pou ek ˆ oun thn katˆstash thc diadikasÐac AutoML.

Pa akˆtw parajètoume ton k¸dika pou perièqei ton o ismì klˆshc kai thn mè odo • kataskeuast tou antikeimènou AutoMLExperiment :

Listing 4.24: H klˆsh AutoMLExperiment

```
1   class AutoMLExperiment ():
```

```
    """A status tracker for the AutoML process."""
    def __init__(self, run_id: str):
        self.automl_orchestrator_run_id = run_id
        try:
            kfputils.get_run(run_id)
        except ApiException as e:
            if e.status == 404:
                raise ValueError("Invalid run ID: '%s'. The run ID you"
                                 " provided does not correspond to a KFP run."
                                 % run_id)
            raise RuntimeError("Failed to retrieve KFP run with ID '%s': %s"
                               % (run_id, str(e)))

        self.mlmd_client = mlmdutils.get_client()
        self.context_type_name = "KfpRun"
        self.context_name = mlmdutils.get_context_name_from_run_id(
            self.automl_orchestrator_run_id)

        self.context = None
        self.automl_configuration_artifacts = []
        self.automl_configuration_run_ids = []

        self._update_all_info()
```

H mè odoc•kataskeuast c thc klˆshc AutoMLExperiment a qikopoieÐ èna antikeÐmeno me mìno to ID tou Eno qhst wt AutoML wc eÐsodo Pa akˆtw aÐnetai h lÐsta me ta gnw Ðsmata enìc antikeimènou 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_con guration_artifacts (list)
- automl_con guration_run_ids (list)

Gia na plh o o sei ton q sth sqetikˆ me thn katˆstash tou pei ˆmatoc AutoML, èna antikeÐmenoAutoMLExperiment pa èqei dÔo asikèc me ìdouc diepa c :

1. summary : Enhme ¸nei ton q sth sqetikˆ me thn katˆstash tou Enorqhstrwt AutoML kai twn Ro¸n Diamìrfwshc , kai gia ta sko met ik c kˆ e miac apì tic Roèc Diamìrfwshc .
2. list_con gurations : Enhme ¸nei ton q sth sqetikˆ me thn arqitektonik tou montèlou kai twn pa amèt wn thc kˆ e diamìrfwshc dioqèteushc pou proteÐnei to auto•sklearn.

Pa akˆtw , a d¸soume mia pio leptomer eikìna twn dÔo aut¸n me ìdwn diepa c . Allˆ p in to kˆnoume autì , a ekjèsoume thn leitourgikìthta mÐac polÔ q simhc mejì-dou, pou onomˆzetai _update_all_info :

Listing 4.25: H oh htik mè odoc _update_all_info thc klˆshc 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 ))
```

Aut h mè odoc è nei ìlec tic plh o o Ðec pou eÐnai sqetikèc me thn die gasÐa AutoML. Autì oh ˆ sthn sunoq anˆmesa se ìlwn twn eid¸n tic plh o o Ðec pou a o oÔn thn die gasÐa. Pio analutikˆ , aut h mè odoc anaktˆ :

- ˆ to MLMD Context tou Eno qhst wt AutoML
- ˆ ta AutoMLCon guration Artifacts
- ˆ ta IDs twn Ro¸n Diamì wshc

4.4.2.1 H Mè odoc list_con gurations

Aut h mè odoc diepa c ek ètei plhroforÐec sqetikèc me tic arqitektonikèc montèlwn twn proteinomènwn diamorf¸sewn dioqèteushc . Autì to kata è nei kal¸ntac thn mè odo _update_all_info pou perigrˆyame nw Ðte a , ètsi ¸ste na sullèxei ìla ta AutoMLCon gura• tion Artifacts (4.4.1.2) pou dhmiou g hkan katˆ to ma get•metalearning•con gurations (4.2.2).

Ac doÔme ton k¸dika thc mejìdou list_con gurations :

Listing 4.26: H mè odoc list_con gurations

```
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      )
```

'Opwc mpo oÔme na doÔme apì ton k¸dika pou ek èsame parapˆnw , h list_con gura• tions tup¸nei ìlec tic plhroforÐec pou eÐnai sqetikèc me ta montèla twn diamorf¸sewn dioqèteushc qrhsimopoi¸ntac thn ohjhtik sunˆ thsh print_suggested_models tou do- mostoiqeÐouml.utils :

Listing 4.27: H ohjhtik sunˆ thsh print_suggested_models apì to domostoiqeÐo 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"   )
```

H akìlou h eikìna eÐnai èna pa ˆdeigma tou pwc h mè odoc list_con gurations tup¸nei ìlec tic plh o o Ðec pou eÐnai sqetikèc me ta montèla twn diamo ¸sewn dioqèteushc ìtan thn kaloÔme sto kelÐ enìc Jupyter Notebook.

4.4.2.2 H Mè odoc summary

H mè odoc summary eÐnai h kÔ ia mè odoc diepa c thc klˆshc AutoMLExperiment pou dÐnei thn dunatìthta ston q sth na pa akolou sei thn katˆstash tou Eno qhst wt AutoML kai twn Ro¸n Diamì wshc , ka ¸c epÐshc kai twn sko met ik c pou kˆ e Ro Diamì wshc pa ˆgei .

P okeimènou na anakt sei autèc tic plhroforÐec , h mè odoc summary qreiˆzetai na anakt sei ta ID twn dioqeteÔsewn pou summetèqoun sthn diergasÐa AutoML tou Kale. 'Eqei

Figure 4.3: Pa ˆdeigma thc exìdou thc me ìdou list_con gurations

dh apokt sei to ID tou Enorqhstrwt AutoML (qarakthristikì automl_orchestrator_run_• id). Epomènwc, gia na anakt sei ta IDs twn Ro¸n Diamìrfwshc , h mè odoc summary kaleÐ thn ohjhtik mè odo _update_all_info thc klˆshc AutoMLExperiment .

Listing 4.28: H mè odoc diepa c summary thc klˆshc AutoMLExperiment

```
def summary (self ):
    """Show a summary of the AutoML process."""
    # Retrieve the current status of the AutoML Orchestrator.
    try :
        status = kfputils .get _run (
            self .automl _orchestrator _run _id ).run .status
    except ApiException as e :
        if e .status == 404:
            raise ValueError ("Invalid run ID: '%s'. The run ID you"
                              " provided does not correspond to a KFP run."
                              % self .automl _orchestrator _run _id )
        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()
```

'Opwc eÐnai ane ì apì to pa apˆnw kommˆti k¸dika , h mè odoc summary tup¸nei , sei iakˆ , tèsse ic dia o etikoÔc tÔpouc plh o o Ðac

1. Thn katˆstash tou Eno qhst wt AutoML
2. 'Ena met ht twn Ro¸n Diamì wshc pou èqoun xekin sei thn ektèlesh touc
3. 'Enan pÐnaka HTML pou metˆei ton a i mì twn Ro¸n Diamì wshc pou èqoun kˆpoia sugkek imènh katˆstash . Gia thn ulopoÐhsh autoÔ , h mè odoc summary q hsimopoieÐ mia ˆllh mè odo thc klˆshc AutoMLExperiment , h opoÐa onomˆetai called _print_con guration_run_status_counts :

Listing 4.29: H mè odoc _print_con guration_run_status_counts thc klˆshc AutoMLExper• iment

```
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"    )
```

Aut h mè odoc arqikˆ tup¸nei ton arijmì twn Ro¸n Diamìrfwshc pou èqoun xekin - sei thn ektèlesh touc , apì ìlec tic proteinìmenec diamorf¸seic dioqèteushc . 'Uste a , tup¸nei ènan pÐnaka pou deÐqnei pìsec apì autèc tic Roèc Diamìrfwshc èqoun mia sugkekrimènh katˆstash , ìpwc gia paˆdeigma : Running , Failed Succeeded . 'Opwc kai h run_automl() (4.1), aut h mè odoc eÐnai tiagmènh epÐshc gia na ekteleÐ- tai mèsa se èna Jupyter Notebook, se peribˆllon enìc Jupyter Server tou Kube ow (2.4.2). An autì sumbaÐnei, tìte o pÐnakac tup¸netai se mo HTML, qrhsimopoi¸n- tac to domostoiqeÐo IPython.core.display [32]. Alli¸c , tup¸netai se mo aploÔ keimènou, qrhsimopoi¸ntac to pakèto tabulate [33] thc Python.

4. 'Enan pÐnaka sÔnoyhc pou pe ièqei thn katˆstash kai ta sko met ik c kˆ e miac Ro c Diamì wshc . Gia na to ulopoi sei autì , h summary q hsimopoieÐ ˆllh mia mè odo thc klˆshc AutoMLExperiment , pou onomˆ etai _print_summary_• table . To akìlou o kommˆti k¸dika deÐqnei ton k¸dika thc oh htik c me ìdou _print_summary_table :

   Listing 4.30: H mè odoc _print_summary_table thc klˆshc AutoMLExperiment

```
    def _print_summary_table(self):
        """Print a summary table for the configuration runs."""
        tabular_data = []

        # This will be the header of the metrics column. When the first
        # (metric_name, score) pair is received for a configuration run,
        # the header will be changed to "Metric (<metric_name>)".
        metric_header = "Metric"

        for i, run_id in enumerate(self.automl_configuration_run_ids):
            # Get the status of the configuration run that has started.
            try:
                status = kfputils.get_run(run_id).run.status
            except ApiException as e:
                if e.status == 404:
                    raise ValueError("Invalid run ID: '%s'."
                                     " The run ID you"
                                     " provided does not"
                                     " correspond to a KFP"
                                     " run." % run_id)
                raise RuntimeError("Failed to retrieve KFP run"
                                   " with ID '%s':"
                                   " %s" % (run_id, str(e)))
            metric = "-"

            # Get the MLMD Context of the configuration run, if exists.
            context_name = mlmdutils.get_context_name_from_run_id(run_id)
            context = self.mlmd_client.get_context_by_type_and_name(
                type_name=mlmdutils.RUN_CONTEXT_TYPE_NAME,
                context_name=context_name)

            if not context:
                tabular_data.append([i + 1, run_id, status, metric])
                continue

            # The Model's ArtifactType may not exist yet and be created
            # during this AutoML experiment.
            try:
                artifacts = \
                    mlmdutils.get_artifacts_by_context_and_type(
                    context_id=context.id,
                    type_name=Model.artifact_type_name)
            except NotFoundError:
                artifacts = []

            if not artifacts:
                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" )
```

Aut h mè odoc tup¸nei ènan pÐnaka pou deÐqnei ton arijmodeÐkth , to ID, thn katˆs-tash kai to sko met ik c gia kˆ e Ro Diamìrfwshc . OmoÐwc me thn pe Ðptwsh thc _print_con guration_run_status_counts() , an h _print_summary_table() ekteleÐtai se Jupyter Notebook, mèsa se peribˆllon Jupyter Server tou Kube ow, tìte o pÐnakac tup¸netai se mo HTML, qrhsimopoi¸ntac to domostoiqeÐo IPython.core.display [32].

H eikìna 4.4 eÐnai èna pa ˆdeigma tou pwc h mè odoc summary tup¸nei tic katastˆseic tou Eno qhst wt AutoML kai twn Ro¸n Diamì wshc , ìpwc epÐshc ta sko met ik c pou kˆ e Ro Diamì wshc pa ˆgei .

Figure 4.4: Pa ˆdeigma kl shc thc me ìdou summary mèsa se èna kelÐJupyter Notebook

Autì pou eÐnai epÐshc shmantikì sqetikˆ me thn ektèlesh thc me ìdou summary mèsa se èna Jupyter Notebook, eÐnai to gegonìc ìti kˆ e ID Ro c Diamì wshc eÐnai sÔndesmoc sthn selÐda thc diepa c q sth tou KFP pou antistoiqeÐ sthn Ro Diamì wshc aut . H eikìna 4.5 deÐqnei thn selÐda thc diepa c q sth tou KFP pou em anÐ etai ìtan kˆnoume klik pˆnw sto p ¸to ID tou pa adeÐgmatoc thc eikìnac 4.4.

Figure 4.5: H selÐda thc diepa c q sth tou KFP pou antistoiqeÐ se mia Ro Diamì wshc

Chapter 5

Axiolìghsh

Se autì to kefˆlaio , a axiolog soume thn apìdosh tou mhqanismoÔ mac kai a ton sugkrÐnoume me thn pio prìsfath èkdosh tou auto•sklearn (aut th stigm h 0.12.7), pou eÐnai h teleutaÐa lèxh thc teqnologÐac sto pedÐo twn ibliojhk¸n pei amˆtwn AutoML. Pio sugkekrimèna , met same thn epÐdosh tou mhqanismoÔ mac kai tou auto•sklearn se ènan arijmì polÔ gnwst¸n set dedomènwn mhqanik c mˆ hshc . Dokimˆsame to mhqanismì mac se set dedomènwn pou qrhsimopoioÔntai gia ergasÐec tìso palindrìmhshc , ìso kai taxinìmhshc , kai ta opoÐa poikÐlloun se mègejoc .

Gia thn axiolìghsh tou mhqanismoÔ mac , sugkrij kame me to mhqanismì tou auto•sklearn tìso me ìso kai qw Ðc to mhqanismì kataskeu c enwmènwn montèlwn pou perilambˆnetai sto auto•sklearn. Ta peirˆmata me kataskeu enwmènwn montèlwn , anˆloga me to set dedomènwn¸ teÐnoun na qreiˆzontai perissìtero q ìno gia na parˆxoun èna apodotikì telikì montèlo apì to mèso peÐ ama Kale•AutoML, to opoÐo katˆ thn axiolìghs mac di kese pe Ðpou t iˆnta leptˆ katˆ mèso ì o .

MeÐname euqaristhmènoi diapist¸nontac ìti o mhqanismìc mac apodÐdei exairetikˆ kalˆ gia ìla ta set dedomènwn pou qrhsimopoi same , kai se kˆpoiec peript¸seic o mhqanismìc mac eÐnai kalÔteroc apì to auto•sklearn kai me kai qw Ðc thn kataskeu enwmènwn montèlwn energopoihmènh.

5.1 E galeÐa , Me odologÐa kai Pe i ˆllon

Ektelèsame ta peirˆmatˆ mac se èna peribˆllon Jupyter Lab miac egkatˆstashc MiniKF (2.4.5), se èna monì kìm o sto Google Cloud Platform [34]. Pio sugkekrimèna , o kìmboc tan exoplismènoc me mÐa CPU 16 pu nwn (Intel Xeon 2.3 GHz • Broadwell) ma Ð me60 GB RAM. O Jupyter Server (2.4.2) pou qrhsimopoi same gia thn ektèlesh twn pei amˆtwn mac p osˆ thse 4 pu nec thc CPU ma Ð me GB RAM kai 4GB apojhkeutikoÔ q¸ ou gia to q¸ o ergasÐac , ta opoÐa sumperˆname ìti tan epa k .

H souÐta axiolìghs c mac apoteleÐtai apì 5 epi lepìmena set dedomènwn mˆ hshc ta opoÐa ek ètoume ston pÐnaka 5.1.

Kˆ e èna apì ta Kale•AutoML peirˆmatˆ mac ektelèsjhke qrhsimopoi¸ntac th sunˆ ths mac run_automl (4.1) tou API, kai qrhsimopoÐhse pènte diamorf¸seic dioqèteushc (number_of_con gurations=5) en¸ to polÔ dÔo diamorf¸seic dioqèteushc mpo oÔsan na ektelestoÔn paˆllhla (max_parallel_con gurations=2). Parìmoia , to peÐ ama Katib

ìnoma	ErgasÐa Mhqanik c Mˆjhshc	AnalogÐa EkpaÐdeushc / Dokim c
MNIST [35]	taxinìmhsh pollapl¸n klˆsewn	80/20
CIFAR-10 [36]	taxinìmhsh pollapl¸n klˆsewn	80/20
GERMAN CREDIT DATA [37]	duadik taxinìmhsh	75/25
WINE QUALITY [38]	palind ìmhsh	75/25
DIABETES [39]	palind ìmhsh	75/25

Table 5.1: Ta set dedomènwn pou qrhsimopoi jhkan gia thn axiolìghsh tou AutoML mhqanismoÔ mac

pou ulopoieÐ to kommˆti eltistopoÐhshc uperparamètrwn thc diergasÐac mac Kale•AutoML ektèlese pènte dokimèc en¸ to polÔ dÔo dokimèc Katib mpo oÔsan na ektelestoÔn pa ˆllhla .

Gia ta antÐstoiqa peirˆmata auto•sklearn, qrhsimopoi same ta antikeÐmena API AutoSklearnClassi er kai AutoSklearnRegressor tou auto•sklearn. Oi diamorf¸seic pa amèt wn eisìdou pou pa eÐqame se autˆ ta antikeÐmena parajètontai ston pÐnaka 5.2.

parˆmetroc	auto-sklearn	auto-sklearn + enwmèno montèlo
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: Oi pa ˆmet oi eisìdou gia ta antikeÐmena AutoSklearnClassi er kai AutoSklearnRegressor pou qrhsimopoi same sta peirˆmatˆ mac me to auto•sklearn

5.2 Apotelèsmata

Pa ousiˆ oume tic metr seic mac ston pÐnaka 5.3. Ektelèsame ìla ta peirˆmata seiriakˆ kai ìqi pa ˆllhla ¸ste na apofÔgoume tuqìn uperbolik q sh thc CPU thc RAM.

Set Dedomènwn	Metrhtikì	auto-sklearn	auto-sklearn + enwmèno montèlo	Kale-AutoML
MNIST	ak Ð eia	0.885	0.942	0.95
CIFAR-10	ak Ð eia	0.25	0.31	0.25
GERMAN CREDIT DATA	ak Ð eia	0.776	0.78	0.78
WINE QUALITY	mèso tet agwnismèno s ˆlma	0.506	0.478	0.457
DIABETES	mèso tet agwnismèno s ˆlma	42.962	42.539	42.902

Table 5.3: Ta apotelèsmata met ik c gia ìla ta pei ˆmata .

MeÐname ikanopoihmènoi lèpontac ìti stic peript¸seic twn set dedomènwn MNIST, GERMAN CREDIT DATA kai WINE QUALITY h diergasÐa mac Kale•AutoML ajmologeÐtai exÐsou kai uyhlìtera apì to mhqanismì tou auto•sklearn . ApodÐdoume aut th eltÐwsh sthn enswmˆtwsh tou mhqanismoÔ mac me to Katib, to opoÐo ekteleÐ eltistopoÐhsh uperparamètrwn sto dh ekpaideumèno montèlo mac me thn uyhlìterh ajmolìghsh .

Qw Đc thn enswmˆtwsh tou Katib, o mhqanismìc mac èqei ta Đdia apotelèsmata me to auto•sklearn qw Đc thn teqnik kataskeu c enwmènwn montèlwn .

Pa ath oÔme ìti , gia thn pe Đptwsh tou set dedomènwn CIFAR•10 , o mhqanismìc kataskeu c enwmènwn montèlwn tou auto•sklearn epideiknÔei shmantik eltĐwsh stic ajmologĐec dokim¸n se sÔgkrish tìso me ton kanonikì mhqanismì metamˆ hshc tou auto•sklearn ìso kai me ton mhqanismì mac . To CIFAR•10 eĐnai to megalÔte o set dedomènwn sth souĐta mac, kai autì od ghse sthn apotuqĐa t i¸n apì tic pènte diamorf¸seic ektèleshc sto Au•toML peĐ amˆ mac. O lìgoc gia thn apotuqĐa tan ìti o klwnopoihmènoc logikìc dĐskoc q¸ ou ergasĐac pou p osˆ thse to Kale se kˆ e ma aut¸n twn Ro¸n Diamìrfwshc den tan epa k c gia na apojhkeÔsei to proepexergasmèno set dedomènwn pou pa ˆq hke apì kˆpoia apì ta mata run•sklearn•transformer (4.3.2).

Chapter 6

Sumpe asmatikˆ Sqìlia

To taxÐdi mac èftase sto tèloc tou . Se autì to kefˆlaio , a epanadiatup¸soume tic suneis- orèc mac kai a sunoyÐsoume ti prosfèrei o mhqanismìc mac . Tèloc , a kleÐsoume aut th diplwmatik ergasÐa ana è ontac ti mellontikì è go mpo eÐ na gÐnei ¸ste na emploutisteÐ o mhqanismìc mac kai na tˆsei sto sÔnolo twn dunatot twn tou .

6.1 Mia Anake alaÐwsh tou MhqanismoÔ mac

Sqediˆsame , ulopoi same kai axiolog same ton mhqanismì AutoML tou Kale. Ac parousiˆ- soume mia anakefalaÐwsh tou ti prosfèrei gia akìma mia o ˆ :

- ˆ EkkineÐ pei ˆmata AutoML sto Kube ow me mÐa mìno kl sh sunˆ thshc tou API.
- ˆ Epit èpei stouc q stec na pa akolou oÔn thn katˆstash tou pei ˆmatoc kai twn met ik¸n apotelesmˆtwn pou pa ˆgei mèsw enìc aploÔ antikeimènou tou API.
- ˆ AxiopoieÐ thn katanemhmènh Ôsh tou Ku e n th kai to mhqanismì eno q st wshc dioqeteÔsewn tou Kale gia na kataneÐmei ta tm mata tou pei ˆmatoc pou mpo oÔn na pa allhlopoih oÔn .
- ˆ Q hsimopoieÐ to mhqanismì metamˆ hshc tou auto•sklearn ¸ste na "warmstart" th diergasÐa kai to mhqanismì eltistopoÐhshc uperparamètrwn tou Katib gia na prosar- mìsei me ak Ð eia to pio apodotikì montèlo .
- ˆ AxiopoieÐ to mhqanismì apìdoshc ekdìsewn se dedomèna tou Kale gia na kˆnei kˆ e èna ekpaideumèno montèlo eÔkola anapa ˆximo kai p os ˆsimo akìma kai metˆ to pè ac ìlou tou pei ˆmatoc .

6.2 Mellontikì 'E go

Mpo oÔme telikˆ na kleÐsoume aut thn ergasÐa me tic mellontikèc kateujÔnseic èreunac tou mhqanismoÔ mac. Sqediˆzoume na tic akolouj soume energˆ katˆ touc epìmenouc m nec qrìnia .

- Epèktash tou antikeimènou mac API AutoMLExperiment (4.4.2) gia na pa akolou eÐ epÐshc thn katˆstash tou Katib Experiment. P oc to pa ìn pa akolou eÐ mìno thn katˆstash twn Eno qhst wt AutoML kai twn Ro¸n Diamì wshc .
- Enswmˆtwsh tou mhqanismoÔ kataskeu c enwmènwn montèlwn tou auto•sklearn ston Kale•AutoML mhqanismì mac gia thn pa agwg akìma isqu ìte wn telik¸n montèlwn .

- Epèktash thc klˆshc Dataset (4.1.1) ¸ste na pe ilam ˆnei èna set axiolìghshc ma Ð me ta set ekpaÐdeushc kai dokim c pou dh pe ièqei .
- Pa oq stouc q stec thc dunatìthtac na pe io Ð oun to q ìno ektèleshc kˆ e Ro c Diamì wshc mèsw tou run_automl API tou Kale, ìpwc ak i ¸c sthn pe Ðptwsh tou 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 preprocess 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 deducted 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 sat• isfactory level can prove to be a di cult 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 e ciently solve the task at hand.

Data scientists face the di cult task of bringing data in a good state and then nding a machine learning model con guration 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•e ective 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 brie y present the most notable existing solutions in the eld of AutoML.

7.3.1 Katib

Katib [4] is Kube ow's tool for hyper•parameter optimization. The main idea be• hind it is to de ne the training procedure inside a container and run this logic multiple times, changing the hyper•parameters of the containerized model each time, via the in• put 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 nd sets of hyper• parameters that satisfy the requirements that the user sets regarding the model's nal performance. Some of these algorithms are: Grid Search and Bayesian Optimization . In addition, Katib supports ENAS and DARTS neural architecture search algorithms for nding neural network architectures that are bound to work well for a given task. For a more detailed overview of all the di erent search algorithms that Katib o ers, refer to the corresponding section in Katib's o cial 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 nd a high•scoring hyper•parameter set. Choosing the correct model for a speci c dataset is already a hard task on its own . In addition.

hyper•parameter optimization is bound to be ine ective, 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]) estima• tor, 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 Kube ow.

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 di erent model con gurations to nd 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 essen• tially 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 unpro• cessed 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 Kube ow , that prevents combining the advantages of both worlds.

Kube ow [3] is a great platform for orchestrating complex work ows on top of Ku• bernetes. 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 com• ponents, Jupyter Notebooks [9], Pipelines [10], rich UIs and more.

In our work, we extended Kale [11], a pipeline orchestration toolkit for Kube ow, 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 Kube ow'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 (classi • cation 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 con gurations that are bound to perform well for the input dataset. These con gurations fully describe entire machine learning pipelines (e.g. data preprocessors and model architecture).
5. For each machine learning con guration, 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 con guration suggests, and producing test scores while the orchestrator monitors them .
7. Once they are all nished, 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 . Neverthe• less, we will provide a su cient 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.

Chapter 8

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 classi cation, 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 scienti c 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 work ow. The pipeline can involve tasks such as:

1. pre•processing
2. feature selection
3. classi cation/regression

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

Scikit•learn's pipeline module o ers a Pipeline [15] class object that chains together scikit•learn transformer and estimator steps into a single ML pipeline. Its main API method, t_transform , sequentially calls the t_transform methods of all the steps that exist in the pipeline.

Scikit•learn uses the term "transformers" to refer to objects that implement data/fea• ture preprocessing in scikit•learn, and o er a t_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 t 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 t_transform method of the Pipeline actually calls the t_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 t_transform method of the Pipeline object calls the t_transform methods of the transformer steps sequentially, and then trains the estimator on the transformed data by calling the t 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 con guration 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 re• sources in a way that di erent sets of processes have access to di erent sets of resources. Some examples of these resources are process IDs , file names and les related to network access.

8.2.3 Containers are not VMs

While both containers and virtual machines essentially are virtualization implemen• tations, they di er signi cantly.

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, di erent 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 les) and container images usually are (this depends on the containerized applica• tion) 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 vir• tualization to deliver software in packages. These packages are essentially containers , or more speci cally: 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 repro• ducibility 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 Kube ow

8.4.1 Overview

Kube ow [3] is a machine learning toolkit for Kubernetes that aims at simplifying scal• ing and deploying ML models to production, by leveraging the advantages of Kubernetes.

Kube ow 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 work ows.

Kube ow is a set of CustomResourceDe nitions 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 Kube ow via these UIs.

In the following sub•sections, we will demonstrate Kube ow's core components.

8.4.2 Jupyter Notebooks

This components is responsible for allowing the user to deploy and manipulate Jupyter Notebooks in their Kube ow 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

Kube ow Pipelines (KFP) [10] is a platform for building and deploying portable, scal• able machine learning work ows based on Docker containers and is one of the Kube ow core components. It's automatically deployed during Kube ow deployment. The Kube ow 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 work ows. It also comes with an SDK for de ning 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 work ow engine. It is an extension to the Kubernetes cluster that makes the execution of work ows possible. The user submits a YAML de nition of a work ow (Work ow CustomResourceDe nition) 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 work ows.

Kube ow Pipelines uses Argo as their work ow engine . The Software Development Kit (SDK) compiles the user's source code into an Argo Work ow CustomResourceDe ni• tion which then must be applied to the cluster.

8.4.4 Katib

Katib [4] is Kube ow's component for hyperparameter optimization of models. The main idea behind Katib is to de ne 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 speci c 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 eld 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 di erent hyper•parameter values as inputs, in search of a set of them that will accomplish a speci ed 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 Custom• Resource 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 nally deciding when to stop the optimization process.

8.4.5 MiniKF

MiniKF [23] is a single•node instance of Kube ow, that can be deployed locally or in the cloud. It combines Kube ow 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 nal 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 work ows. 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 o cial 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

```
@step(name="my_step")
def step_1(in_1, in_2):
    # 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

```
class ExampleStep(Step):
    name = "example-step"
    def do_run(self, param1, param2):
        # 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

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

```
4         if param   == "do" :
5             res2  = step _2(res1 )
```

8.5.4 Kale's Domain Speci c Language

Kale•SDK's domain speci c language is meant to provide a Python•like API to de ne a pipeline. From this point and on, we will use the acronym: "DSL", to refer to the term: "domain speci c 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 com• parison 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 work ows and experiments. From this point on• wards, we will refer to the Machine Learning Meta•Data store as "MLMD".

MLMD de nes entities, that can be stored and retrieved to/from the database. In our work, we associate each one of these entities with a speci c concept of a ML work ow 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 de ne a "context". We map these entities to KFP Runs.
- Executions : An Execution corresponds to something that executes/runs. An Exe• cution 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 di erent 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

Chapter 9

Our Study of Auto•sklearn 's Meta•Learning Mech• anism

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 leverag• ing recent advantages in Bayesian optimization, meta•learning and ensemble construc• tion, it manages to completely replace any scikit•learn ([2]) estimator, for supervised ma• chine 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 mech• anism. 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 vec• tors that are stored in its meta•learning database and it nds the most similar one. Essentially, each of these other meta•feature vectors corresponds to a speci c dataset, so in other words auto•sklearn nds the most similar dataset that exists in its meta•learning database.
3. suggests a set of machine learning pipeline con gurations that scored highly on the most similar dataset. auto•sklearn keeps these con gurations in its meta•base as well, along with their respective metric scores for each dataset. These suggested con gurations are bound to work well on the input dataset.

9.2 Machine Learning Pipeline Con gurations

9.2.1 Overview

A machine learning pipeline con guration 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 con gurations. We will describe this meta•learning database in a following section.

auto•sklearn 's goal is to suggest con gurations that are bound to score highly for a given machine learning dataset and task. The snippet below shows an example meta• learning con guration for a classi cation task:

Listing 9.1: A meta•learning con guration for classi cation

```
balancing :strategy , Value : 'weighting'
classifier :__choice __, Value : 'libsvm _svc'
classifier :libsvm _svc :C, Value : 6384.641073379224
classifier :libsvm _svc :coef0 , Value : •0.1592835134753816
classifier :libsvm _svc :degree , Value : 2
classifier :libsvm _svc :gamma, Value : 0.6866143858851854
classifier :libsvm _svc :kernel , Value : 'poly'
classifier :libsvm _svc :max_iter , Constant : •1
classifier :libsvm _svc :shrinking , Value : 'False'
classifier :libsvm _svc :tol , Value : 2.6500330000385803 e•05
data _preprocessing :categorical _transformer :categorical _encoding :__choice __, Value : 'no _encoding'
data _preprocessing :categorical _transformer :category _coalescence :__choice __, Value : '
    no_coalescense'
data _preprocessing :numerical _transformer :imputation :strategy , Value : 'median'
data _preprocessing :numerical _transformer :rescaling :__choice __, Value : 'normalize'
feature _preprocessor :__choice __, Value : 'no _preprocessing'
```

Similarly, the following snippet shows a real meta•learning con guration that could be used for a regression task:

Listing 9.2: A meta•learning con guration for regression

```
data _preprocessing :categorical _transformer :categorical _encoding :__choice __, Value : 'no _encoding'
data _preprocessing :categorical _transformer :category _coalescence :__choice __, Value : '
    no_coalescense'
data _preprocessing :numerical _transformer :imputation :strategy , Value : 'most _frequent'
data _preprocessing :numerical _transformer :rescaling :__choice __, Value : 'robust _scaler'
data _preprocessing :numerical _transformer :rescaling :robust _scaler :q_max, Value :
    0.8280417820125114
data _preprocessing :numerical _transformer :rescaling :robust _scaler :q_min , Value :
    0.0781634653874277
feature _preprocessor :__choice __, Value : 'kitchen _sinks'
feature _preprocessor :kitchen _sinks :gamma, Value : 8.438432240830361 e•05
feature _preprocessor :kitchen _sinks :n_components , Value : 2984
regressor :__choice __, Value : 'ard _regression'
regressor :ard _regression :alpha _1, Value : 0.00044036509169446026
regressor :ard _regression :alpha _2, Value : 4.039147500668822 e•10
regressor :ard _regression :fit _intercept , Constant : 'True'
regressor :ard _regression :lambda _1, Value : 8.922721154590444 e•05
regressor :ard _regression :lambda _2, Value : 3.0105431227198885 e•05
regressor :ard _regression :n_iter , Constant : 300
regressor :ard _regression :threshold _lambda , Value : 1899.168836704701
```

```
18 regressor :ard _regression :tol , Value : 0.011611373742389547
```

As it can be seen from above, a con guration is a dictionary•like object that de nes 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 Con guration Space

The con guration space essentially consists of all the pipeline con gurations that are valid for a given ML task. For example, for the case of a classi cation task, the con gu• ration space will contain pipeline con gurations that only have classi ers as estimators. Similarly, for the case of a regression task, the con guration space will contain pipeline con gurations that only have regressors .

However, the type of the given ML task is not the only criterion that a ects the struc• ture of the con guration space. More speci cally, the structure of the con guration space is a ected 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 con guration space is one of the rst things that auto•sklearn 's meta•learning mechanism calculates, before producing the suggested pipeline con gurations.

9.2.2.1 The get_con guration_space API Function

The auto•sklearn package exposes the get_con guration_space API function which produces a con guration space object for a given ML task. Here's the signature of the get_con guration_space API function:

ˆ Input Arguments :

info : A dictionary that contains the information that is necessary to compute the con guration 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 in nite values in the dataset.

include_estimators : A list with the names of the estimators to be used, ex• clusively. 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" [1].

ˆ 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

```
class XYDataManager(AbstractDataManager):

    def __init__(
        self,
        X: np.ndarray,
        y: np.ndarray,
        X_test: Optional[np.ndarray],
        y_test: Optional[np.ndarray],
        task: int,
        feat_type: List[str],
        dataset_name: str
    ):
        super(XYDataManager, self).__init__(dataset_name)

        self.info['task'] = task
        if sparse.issparse(X):
            self.info['is_sparse'] = 1
            self.info['has_missing'] = np.all(np.isfinite(X.data))
        else:
            self.info['is_sparse'] = 0
            self.info['has_missing'] = np.all(np.isfinite(X))

        label_num = {
            REGRESSION: 1,
            BINARY_CLASSIFICATION: 2,
            MULTIOUTPUT_REGRESSION: y.shape[-1],
            MULTICLASS_CLASSIFICATION: len(np.unique(y)),
            MULTILABEL_CLASSIFICATION: y.shape[-1]
        }

        self.info['label_num'] = label_num[task]

        self.data['X_train'] = X
        self.data['Y_train'] = y
        if X_test is not None:
            self.data['X_test'] = X_test
        if y_test is not None:
            self.data['Y_test'] = y_test

```

[1] The value of a categorical feature states that a sample belongs to a certain category. These type of features have a nite 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          if self.feat_type is None:
60              self.feat_type = ['Numerical'] * X.shape[1]
61          if X.shape[1] != len(self.feat_type):
62              raise ValueError('X and feat_type must have the same number of columns, '
63                               'but are %d and %d.' %
64                               (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 SimpleClassi cationPipeline Classes

These two classes implement the classi cation 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 con guration object (9.2) to the constructor of the corresponding class. After that, one can call the:

1. t method of the object to t 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 cate• gories:

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 ex• pensive, 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 [2]
- 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 speci cally, 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

[2] 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 cal•culate 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 parame•ters 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 speci c 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 speci cally, 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 met•alearning.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 con gurations 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 con gurations for that speci c ML task.

To construct a MetaBase object, we need to provide:

- a con guration 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 cor• responding dataset

Adding an input dataset in the MetaBase object is essential to calculating the sug• gested con gurations 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 mod• ule is one of the most important API functions in auto•sklearn 's meta•learning mecha• nism, since it is the one that actually produces the suggested pipeline con gurations 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 direc• tory and the con guration space in which suggest_via_metalearning will search for the suggested con gurations.

  dataset_name : The name of the dataset that the suggested con gurations 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_con gurations : The number of suggested con gurations to pro• duce.

- Return Value :

  con gurations : A list of the suggested con gurations.

In a following chapter, we will describe how Kale's ML back•end uses the suggest_• via_metalearning to produce the suggested con gurations 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 re• quirements. 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_classi cation : A bool value that expresses whether the task that the input dataset is going to be used for is classi cation 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_classi cation input argument, it checks that the targets in the dataset can be used for that task (classi cation 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 con gurations, 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 e ciently 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 (classi - cation 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 con gurations for the user's input dataset and task. Each of these con gurations essentially describes an entire machine learning pipeline .
5. For each suggested pipeline con guration, the AutoML Orchestrator spins up a new KubeFlow pipeline . These pipelines are called Con guration Runs and we will be referring to them by that name for the rest of the chapter. Each one of these Con- guration Runs implements the ML pipeline that the corresponding con guration suggests.
6. Con guration Runs run in parallel , preprocessing the dataset, training the model, and producing test scores while the AutoML Orchestrator monitors them .
7. Once all Con guration Runs are nished, the AutoML Orchestrator gathers their scores and selects the best Con guration Run .

8. The AutoML Orchestrator creates a Katib experiment to further optimize the trained model of the best scoring Con guration 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 Con guration 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 . Neverthe• less, we will provide a su cient 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

```
def run_automl(
    dataset: Dataset, task: MLTask, metric: Callable,
    number_of_configurations: int = 5,
    max_parallel_configurations: int = 3,
    tuner: Optional[katib.V1beta1ExperimentSpec] = None
) -> AutoMLExperiment:
    """Runs an AutoML pipeline to find the best model for the input dataset.

    [... Explain how the AutoML process works ...]

    Args:
        dataset (common.artifacts.Dataset): The input dataset for the ML task
        task (types.MLTask): One of kale.ml.Task
        metric (Callable): A callable object with the following call signature

            >>> class my_metric:
            >>>     def __call__(self, target, x, test):
            >>>         return self._compute_my_metric_value(target, x, test)

            The name of the logged metric will be `metric.name`, if the object
            has such attribute. Otherwise ``metric.__name__``.
            (Auto)SKLearn metrics are supported, example:

            >>> from autosklearn.metrics import accuracy

            To log a different metric name from the input function name
            (``foo.__name__``), do:

            >>> foo.name = "<custom_name>"

        number_of_configurations (int): The N-best configurations to run
            (defaults to 5)
        max_parallel_configurations (int): The maximum number of Configuration
            Runs to run in parallel (defaults to 3)
        tuner (katib.V1beta1ExperimentSpec): Provide a Katib spec to run HP
            Tuning over the best performing configuration.

            Cannot set algorithm and parameters. To set objective
            configuration, don't set metric name:

```

```
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 classi cation
2. multi•class classi cation
3. simple regression

- metric : A callable metric function that will be used to evaluate the trained model produced by each Con guration Run.
- number_of_con gurations : The number of suggested con gurations that the AutoML Orchestrator will extract. The AutoML Orchestrator will eventually create a Con guration Run for each of these con gurations.
- max_parallel_con gurations : The maximum number of Con guration 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 Con guration Runs.
- tuner : A Katib experiment speci cation 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 rst analyzed the functionality behind the AutoML Orchestrator pipeline (10.2) and the Con guration Runs (10.3).

Here is a numbered list of steps that describes how run_automl() API function manages to touch o 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 Con guration Runs can mount the cloned volume to nd and load the parameters.
2. It instantiates a kale.PipelineCon g 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 Work ow 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 Con guration Runs and also to view the metric scores of the Con guration 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

```
def automl_orchestrate(hp_tune="false"):
    """Auto ML pipeline."""
    (configurations,
     kale_dataset_id) = GetMetaLearningConfigurations()(
        ml_assets_marshal_path("dataset.dillpkl"),
        ml_assets_marshal_path("task.dillpkl"),
```

```
        ml_assets_marshal_path("metric.joblib"),
        ml_assets_marshal_path("number_of_configurations.dillpkl"))

    run_ids = RunMetaLearningConfigurations()(
        configurations,
        ml_assets_marshal_path("max_parallel_configurations.dillpkl"),
        kale_dataset_id)
    run_ids = MonitorKFPRuns()(run_ids)
    best_configuration = GetBestConfiguration()(
        run_ids,
        configurations,
        ml_assets_marshal_path("metric.joblib"))
    if hp_tune == "true":
        katib_experiment_name = RunKatibExperiment()(
            ml_assets_marshal_path("tuner.dillpkl"),
            best_configuration,
            ml_assets_marshal_path("metric.joblib"),
            kale_dataset_id)
        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. GetMetaLearningCon gurations
2. RunMetaLearningCon gurations
3. MonitorKFPRuns
4. GetBestCon guration
5. RunKatibExperiment
6. MonitorKatibExperiment

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

10.2.2 The GetMetaLearningCon gurations Step

In this subsection, we will expose and explain the mechanism that the get•metalearning•con gurations step implements. We will describe how this step uses auto•sklearn 's meta•learning kernel to generate a list of candidate pipeline con gurations that are likely to perform well for a given ML task. Let's view the class de nition of the get•metalearning•con gurations 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•con gurations step

```
class GetMetaLearningConfigurations(Step):
    """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 con gurations to be returned.

- Return Value :

  con gurations : A list of pipeline con gurations (section 9.2).

The procedure of computing the pipeline con gurations 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_con gs :

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 in• formation about our experiment, such as whether the input dataset is sparse or not. This information is essential for Kale to calculate the con guration space (sub• section 9.2.2) which is essential for auto•sklearn to suggest accurate meta•learning con gurations, later on in the process.
4. It nds the metadata directory (subsection 9.2.3) that corresponds to the combina• tion of:
   - the type of the metric
   - the type of the ML task
   - whether the dataset is sparse or not.

   To do this, compute_con gs calls nd_metadata_dir , a utility function from Kale 's ml.utils module. We will expose and describe this function in a following section.
5. It nds the con guration space (subsection 9.2.2) in which auto•sklearn will search for pipeline con gurations. For this purpose, compute_con gs calls get_con gura• tion_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 con gurations (section 9.2).

Here is the code that runs inside compute_con gs :

Listing 10.5: The compute_con gs function that produces a list of suggested pipeline con gurations

```
def compute_configs (dataset : Dataset ,
                     task : MLTask,
                     metric : Callable ,
                     number_of_configurations : int ) -> List [Configuration ]:
    """Use the AutoSKLearn MetaLearning system to produce ML configurations.

    The AutoSKLearn MetaLearning system is based on prior knowledge on how
    certain Machine Learning models perform on a set of known datasets.
    AutoSKLearn can use this prior knowledge to suggest some Machine Learning
    configurations that are supposed to perform well on a new, previously
    unseen, dataset."""
    log.info ("Getting suggested configurations..." )

    task = mltask_to_string (task )

    validated_dataset , feature_types = _validate_dataset (dataset , task )
    task_type = extract_task_type (y=validated_dataset .targets , task =task )

    metafeatures = calculate_all_metafeatures (x=validated_dataset .features ,
                                               y=validated_dataset .targets ,
                                               dataset_name=dataset .name,
                                               task_type =task_type ,
                                               feature_types =feature_types )

    # XYDataManager does some validation to the dataset and the list of feature
    # types. It also finds if the dataset is sparse or not - useful for
    # detecting the metadata directory.
    datamanager = XYDataManager (X=validated_dataset .features ,
                                 y=validated_dataset .targets ,
                                 X_test =validated_dataset .features_test ,
                                 y_test =validated_dataset .targets_test ,
                                 task =task_type ,
                                 feat_type =feature_types ,
                                 dataset_name=dataset .name)
    is_sparse = datamanager .info ["is_sparse" ]

    metadata_directory = find_metadata_dir (task_type , metric , is_sparse )
    config_space = get_configuration_space (datamanager .info )
    # The MetaBase object is a container for metafeatures, configurations
    # and their respective scores.
    meta_base = MetaBase (config_space , metadata_directory )
    meta_base .add_dataset (dataset .name, metafeatures )
    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 specically 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.

Tho 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 nd_metadata_dir Function

The nd_metadata_dir function, as its name suggests, nds 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 nd_metadata_dir function has the following signature:

- Input Arguments :

  task_type : A kale.types.MLTask that describes the type of the machine learn• ing task.
  metric : The metric function that is used to evaluate the results of the experi• ment.
  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 nd_metadata_dir :

Listing 10.7: The compute_con gs function that nds the path to the meta•data directory that corresponds to a given ML task

```
def find _metadata _dir (task _type : int , metric : Callable , is _sparse : int ):
    """Find the directory where auto-sklearn stores its meta-knowledge.

    Note:
        The directory structure follows the 'Algorithm Selection Library'
        (ASLib) format. See https://www.automl.org/automated-algorithm-design/algorithm-
      selection/aslib/ # noqa: 501

    Args:
        task _type (int): The type of the ML task.
        metric (callable): The metric that is used for the ML task.
        is _sparse (int): Whether the dataset is sparse or not.

    Returns:
        str: path to the auto-sklearn metadata directory.

    Raises:
        RuntimeError: When the auto-sklearn metadata directory cannot be found.
    """
    log .info ("Finding metadata dir..." )
    metalearning _directory = os.path .dirname (autosklearn .metalearning .__file __)
    # The auto-sklearn metadata directory doesn't provide metadata for
    # multi-label classification. auto-sklearn reverts to using binary
    # classification as well, so we copy this behaviour.
    if task _type == constants .MULTILABEL_CLASSIFICATION :
        meta _task = constants .BINARY_CLASSIFICATION
    else :
        meta _task = task _type
    metalearning _files _dir = "%s_%s_%s" % (
        metric , constants .TASK_TYPES_TO_STRING[meta _task ],
        "sparse" if is _sparse else "dense" )
    metadata _directory = os.path .join (
        metalearning _directory , "files" , metalearning _files _dir )
    if not os .path .exists (metadata _directory ):
        raise RuntimeError ("Metadata directory %s does not exist."
                            % metadata _directory )
    log .info ("Metadata directory: %s" , metadata _directory )
    return metadata _directory
```

10.2.2.4 The _submit_con guration_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_con guration_artifact method creates an AutoMLCon guration Artifact (subsubsection 10.4.1.2) out of a given

pipeline con guration.

Listing 10.8: The _submit_con guration_artifact method of GetMetaLearningCon gura• tions that creates and submits an AutoMLCon guration Artifact for a given pipeline con• guration

```
def _submit_configuration_artifact(self, configuration, idx):
    from kale.ml import utils
    from kale.common.artifacts import AutoMLConfiguration

    mlmd = mlmdutils.get_mlmd_instance()
    config_summary = utils.get_configuration_summary(configuration)
    config = AutoMLConfiguration(
        config_summary=config_summary,
        run_id=kfputils.format_kfp_run_id_uri(mlmd.run_uuid),
        estimator_name="Configuration %s: %s"
                       % (idx + 1, config_summary["name"]))
    config.assign_list_index(idx + 1)
    config_artifact = config.submit_artifact()
    mlmd.link_artifact_as_output(config_artifact.id)
```

As we can see, the AutoMLCon guration class expects a con g_summary input pa• rameter which describes all the important information about the estimator (classi er, or regressor) of the pipeline con guration. More speci cally, con g_summary is expected to be a dictionary that describes the estimator with the following elds:

- 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 con g_summary parameter, this step uses the get_con guration_sum• mary utility function from Kale's ml.utils module:

Listing 10.9: The get_con guration_summary utility function that creates a summary dictionary out of a given pipeline con guration

```
def get_configuration_summary(configuration: Configuration) -> Dict[str, Any]:
"""Return an opinionated summary of the input configuration.

The output of this function can be used to pretty-print a configuration,
with just the right information, or to upload the configuration to the
artifact store.

Args:
    configuration: A suggested configuration extracted by auto-sklearn.

Returns:
    dict: A dictionary that describes the learner (classifier, or
        regressor) of the configuration, with the following fields:

        * ``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."""
```

```
        from time import sleep
        from kale import marshal
        from kale.common import mlmdutils
        from kale.ml.utils import ML_ASSETS_DIR
        from kale.common.artifacts import AutoMLConfiguration

        self.vars["run_ids"] = []
        marshal.set_data_dir(ML_ASSETS_DIR)

        mlmd = mlmdutils.get_mlmd_instance()
        # Get all Artifacts that are attributed to the Context of the AutoML
        # Orchestrator. The SKLearnTransformer step of each new Configuration
        # Run should link the corresponding AutoMLConfiguration Artifact as its
        # input. So, we should pass the corresponding AutoMLConfiguration
        # Artifact ID to each Configuration Run.
        kale_config_artifacts = mlmdutils.get_artifacts_by_context_and_type(
            context_id=mlmd.run_context.id,
            type_name=AutoMLConfiguration.artifact_type_name,
            sorted=True)
        if len(kale_config_artifacts) != len(configurations):
            raise RuntimeError("Founds %d MLMD configuration artifacts but"
                               " %d configuration were provided as input."
                               % (len(kale_config_artifacts),
                                  len(configurations)))

        # Start all configurations with a reconciliation loop to avoid having
        # more than max_parallel_configurations running concurrently
        while configurations:
            if self._running_ids() >= max_parallel_configurations:
                log.info("Cannot start a new configuration. Max parallel"
                         " configurations cap is set to %d. Waiting for a"
                         " configuration to complete...",
                         max_parallel_configurations)
                sleep(10)
                continue

            configuration = configurations.pop(0)
            index = len(self.vars["run_ids"]) + 1

            log.info("Saving configuration n. %d", index)
            marshal.save(configuration, "configuration")

            automl_config_artifact_id = kale_config_artifacts[index - 1].id
            run_id = self._run_pipeline(index, {
                "kale_dataset_id": str(kale_dataset_id),
                "kale_config_id": str(automl_config_artifact_id)})
            self.vars["run_ids"].append(run_id)
            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 Con• figuration Run will mark it as its input, sustaining a MLMD lineage for the experiment.
  - max_parallel_configurations: An integer that expresses the maximum num• ber 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 Con guration Run ID in the list that holds all Con guration Run IDs . This list is the only output of the RunMetaLearningCon gurations step. It will be used by the next step (MonitorKFPRuns)to query the KFP server and monitor the status of each individual Con guration Run.
6. Update the MLMD Context of the Orchestrator with links to the Contexts of the Con guration Runs. This way, we sustain a MLMD lineage between the AutoML Orchestrator and the Con guration Runs.

10.2.3.1 The _run_pipeline Method

As we described above, when the number of running Con guration Runs is less than the value of max_parallel_con gurations (input parameter in the run_automl API function), the RunMetaLearningCon gurations step picks a con guration and spins up a new Con guration Run. The creation of the Con guration Run is implemented by the _run_pipeline method of the RunMetaLearningCon gurations class. The following list describes the signature of the method:

- Input Arguments :

  index : An integer that ranges from 1 to number_of_con gurations . This integer is unique for the experiment, and it corresponds to the position of the con guration in the list of extracted con gurations.

  params : A dictionary that contains the following elds:
  * kale_dataset_id : The ID of the Kale Dataset Artifact (10.4.1.3).
  * kale_con g_id : The ID of the AutoMLCon guration Artifact (subsection 10.4.1.2) which corresponds to the Con guration Run and which describes the corresponding pipeline con guration. To sustain a MLMD lineage for the AutoML experiment, we pass the AutoMLCon guration Artifact ID to each individual Con guration 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 Con guration Run with the Artifact. Linking an AutoMLCon guration Artifact as input expresses the fact that the pipeline receives the corresponding con guration as its input from the AutoML Orchestrator.

  These elds are to be added to the pipeline parameters of the newly created Con guration Run.

- Return Value :

  the run ID of the Con guration Run.

Listing 10.11: The _run_pipeline method of the RunMetaLearningCon gurations step class

```
def _run_pipeline(self, index, params: Dict = {}):
    from kale.types import Param
    from kale import PipelineConfig, Compiler
    from kale.processors import PythonProcessor
    from kale.ml.pipelines import sklearn_train_predict

```

```
        volumes = rokutils.interactive_snapshot_and_get_volumes()
        pipeline_config = PipelineConfig(
            pipeline_name="sklearn-configuration-%d" % index,
            experiment_name=kfputils.get_experiment_from_run_id(
                kfputils.detect_run_uuid()).name,
            marshal_path=self.marshal_path,
            volumes=volumes)
        pipeline_params = {k: Param(type(v).__name__, v)
                           for k, v in params.items()}

        log.newline()
        log.info("Creating pipeline for configuration n. %d", index)
        processor = PythonProcessor(
            sklearn_train_predict.sklearn_train_predict, pipeline_config)
        processor.pipeline.default_pipeline_parameters.update(pipeline_params)
        pipeline = processor.run()
        log.info("Running pipeline for configuration n. %d", index)
        run = Compiler(pipeline).compile_and_run()
        log.info("Successfully started run %s" % run.id)
        log.newline()
        return run.id
```

As we described above, _run_pipeline essentially creates actual KubeFlow Pipelines, called Con guration Runs. For this purpose this, this method mainly uses Kale's back• end (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 me mounted to the rst step of each Con guration Run, and it contains a marshal object with the corresponding pipeline con guration.
2. It instantiates a kale.PipelineCon g object with all the basic descriptive infor• mation (such as the name) of the Con guration Run.
3. It creates a dictionary with kale_dataset_id and kale_con g_id Param objects that will be passed as pipeline parameters to the Con guration 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 Con guration 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 Work ow and applies it to Ku• bernetes. 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 MonitorKF• PRuns step implements. In summary, this step watches the statuses of the Con guration

Runs that were created by the previous step (run•metalearning•con gurations), and when they are all nished, either successfully or unsuccessfully, this step completes its execu• tion. Let us view the code that runs inside this Kale step, and more speci cally its class de nition and do_run method:

Listing 10.12: The class de nition and do_run() method of the monitor•kfp•runs step

```
class MonitorKFPRuns (Step):
    """Wait for KFP pipelines to complete.

    Ins:
        run_ids

    Outs:
        run_ids
    """
    name = "monitor-kfp-runs"
    outs = Param.odict (["run_ids"], step_name=name)

    def do_run (self, run_ids):
        """Implementation of MonitorKFPRuns."""
        from time import sleep

        log.info ("Monitoring runs: %s", run_ids)
        statuses = {run_id: "Pending" for run_id in run_ids}

        # Add custom properties linking the MLMD Execution with the runs to
        # monitor
        log.info ("Patching MLMD Execution custom properties with the"
                  " configuration run IDs...")
        mlmd = mlmdutils.get_mlmd_instance ()
        custom_props = {"configuration_run_%d" % idx:
                        kfputils.format_kfp_run_id_uri (run_ids[idx])
                        for idx in range (len (run_ids))}
        mlmdutils.patch_execution_custom_properties (mlmd.execution.id,
                                                     custom_props)
        log.info ("Successfully patched MLMD Execution")

        while any (map(lambda state: state not in kfputils.KFP_RUN_FINAL_STATES,
                       statuses.values ())):
            log.newline ()
            log.info ("Updating pipelines statuses...")
            for run_id in statuses.keys ():
                if statuses[run_id] not in kfputils.KFP_RUN_FINAL_STATES:
                    statuses[run_id] = kfputils.get_run (run_id).run.status
                    log.info ("Run '%s': %s", run_id, statuses[run_id])
            sleep (5)

        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"
```

```
        outs = Param.odict([("best_configuration", )], step_name=name)

    def do_run(self, run_ids, configurations, metric):
        """Implementation of GetBestConfiguration."""
        metrics = dict()
        for run_id in run_ids:
            log.info("Collecting metrics for run: %s", run_id)
            metrics[run_id] = kfputils.get_kfp_run_metrics(run_id)
        final_metrics = metrics.copy()

        log.newline()
        log.info("Collected metrics:\n")
        for run_id, _metrics in metrics.items():
            log.info("  Run %s:", run_id)
            if not _metrics.values():
                log.info("    No metrics found.")
                del final_metrics[run_id]
            for name, value in _metrics.items():
                log.info("    %s: %s", name, value)
        log.info("Using metric '%s' as target metric.", metric.name)

        # Get best metric, excluding empty metrics dictionaries
        opt = max if metric._sign == 1 else min  # see arrikto/dev#1128
        best_run_uid, best_metric = opt(final_metrics.items(),
                                        key=lambda x: x[1][metric.name])
        log.info("Best run id: %s", best_run_uid)

        log.info("Patching MLMD Execution and Context custom properties with"
                 " the best configuration run ID...")
        custom_prop = {"best_configuration_run": :
                       kfputils.format_kfp_run_id_uri(best_run_uid)}
        mlmd = mlmdutils.get_mlmd_instance()
        mlmdutils.patch_execution_custom_properties(mlmd.execution.id,
                                                    custom_prop)
        log.info("Successfully patched MLMD Execution")
        mlmdutils.patch_context_custom_properties(mlmd.run_context.id,
                                                  custom_prop)
        log.info("Successfully patched MLMD Context")

        best_configuration = configurations[run_ids.index(best_run_uid)]
        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.

con gurations : A list of the suggested pipeline con gurations. This is actually the output of the GetMetaLearningCon gurations step. The GetBestCon gura• tion step extracts and returns the best pipeline con guration out of that list.

metric : The metric that is used to evaluate the results of each Con guration Run. This is actually the function object that the user provided as input to the run_automl API function (section 10.1).

ˆ Return Value :

con guration : The highest scoring con guration in the AutoML process.

Here is a numbered list of steps that describes the mechanism behind GetBestCon• guration 's do_run method:

1. Collect the metric scores of the Con guration 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 Con guration Run (and con guration) 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 Con guration Run . As we mentioned in section 8.6, executions are mapped to pipeline steps. Therefore, since the GetBestCon guration step nds the highest scoring Con guration Run, we link this step to that Con guration 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 con guration that corresponds to the best scoring Con guration 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 con guration from the previous step, and creates a Katib experiment (8.4.4) to further optimize this top• scoring con guration by nding hyper•parameter values that produce even better results than the starting con guration.

This and the following step (monitor•katib•experiment) are executed only in the case that the user has speci ed 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•con guration 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 de nition and the main methods of the step:

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

```
class RunKatibExperiment(Step):
    """Run a Katib experiment.

    Ins:
        tuner (katib.V1beta1ExperimentSpec): Experiment spec
        best_configuration (ConfigSpace.Configuration):
        metric (autosklearn.metric): An (Auto)SKLearn metric
        kale_dataset_id (int): The MLMD artifact ID

    Outs:
        katib_experiment_name (str): Katib experiment name
    """
    name = "run-katib-experiment"
    outs = Param.odict(["katib_experiment_name"], step_name=name)
    actions = ["KatibExperiment"]

    def _get_hyperparams(self, configuration):
        return {
            hp_name: configuration[hp_name]
            for hp_name in configuration.keys()
            if (any(map(lambda alg_type: hp_name.startswith(alg_type),
                        ["classifier", "regressor"]))
                and not hp_name.endswith("__choice__"))}

    def _generate_hyperparam_conf(self, key, value, conf_space):
        conf_space = copy.deepcopy(conf_space)
        hp = conf_space.get_hyperparameter(key)
        if hasattr(hp, "upper") or hasattr(hp, "choices"):
            hp.default_value = value
            return hp
        return None

    def _get_hyperparams_confs(self, configuration):
        configuration = copy.deepcopy(configuration)
        hyperparams_conf = []
        for key, value in self._get_hyperparams(configuration).items():
            hp = self._generate_hyperparam_conf(
                key, value, configuration.configuration_space)
            if hp:
                hyperparams_conf.append(hp)
        return hyperparams_conf

    def _get_param(self, conf):
        from kubeflow import katib
        if hasattr(conf, "choices"):
            return katib.V1beta1ParameterSpec(
                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 con guration 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 speci ed 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 di erent hyper•parameter values to gure out how the hyper•parameters a ect 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 de nition 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
```

```
    def do_run(self, katib_experiment_name: str):
        """Implementation of MonitorKatibExperiment."""
        from kale.common import katibutils

        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

```
def wait_for_hptuning_experiment(experiment_name: str):
    """Wait for an HP Tuning experiment to succeed.

    Args:
        experiment_name (str): Name of the HP Tuning experiment.

    Returns:
        tuple(str, str, str): Status, Condition reason, Condition message.
    """
    def sleep_with_progress(total, interval, msg):
        for i in range(0, total // interval):
            log.info("%s %d..." % (msg, total - i * interval))
            time.sleep(interval)

    while True:
        log.newline(2)
        log.info("Watching for HP Tuning experiment: '%s'",
                 experiment_name)
        sleep_with_progress(30, 5, "Checking status in")

        experiment = get_experiment(experiment_name, podutils.get_namespace())
        status = get_experiment_status(experiment["status"])
        log.info("Experiment status: %s", status)
        if status[0] not in EXPERIMENT_FINAL_STATES:
            continue
        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

speci c part of a ML work ow. Here's an overview of each step:

1. run•sklearn•transformer : This step implements the preprocessor speci ed in a pipeline con guration. 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 e ciently be trained and tested on it.
2. train•sklearn•estimator : This step essentially implements the actual model archi• tecture that a con guration 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 nally produced, the entire Con guration Run nishes its execution.

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

Figure 10.2: Example of completed Con guration 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 Con guration 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•speci c language (8.5.4), that essentially describes the structure of the Con g•uration Run that is created by the RunMetaLearningCon gurations (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

```
def sklearn_train_predict(kale_dataset_id="-1", kale_config_id="-1"):
    """Train and validate a SKLearn model from an AutoML configuration."""
    (x_processed,
     x_test_processed,
     kale_dataset_id_local,
     transformer_pipeline) = RunSKLearnTransformer()(
        ml_assets_marshal_path("configuration.dillpkl"),
        ml_assets_marshal_path("dataset.dillpkl"),
        kale_dataset_id,
        kale_config_id,
        "{}"
    )
    model, kale_model_id = TrainSKLearnEstimator()(
        ml_assets_marshal_path("configuration.dillpkl"),
        x_processed,
        ml_assets_marshal_path("dataset.dillpkl"),
        kale_dataset_id_local,
        "{}"
    )
    InferSKLearnPredictor()(model,
                            x_processed,
                            x_test_processed,
                            ml_assets_marshal_path("metric.joblib"),
                            ml_assets_marshal_path("dataset.dillpkl"),
                            ml_assets_marshal_path("task.dillpkl"),
                            kale_model_id,
                            kale_dataset_id_local)
```

As we see from the previous code snippet, the Con guration 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 speci c part of the Con guration 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 con guration and implements the preprocessor step that this con guration describes. Let us view the class de nition and the main methods of the step:

Listing 10.18: The class de nition and the main methods of the run•sklearn•transformer step

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

    Ins:
        configuration
        dataset
        kale_dataset_id
        kale_config_id

    Outs:
        x_processed
        x_test_processed
        kale_dataset_id_local
        transformer_pipeline

    MLMD Inputs:
        kale.Configuration
        kale.Dataset

    MLMD Outputs:
        kale.Transformer
    """
    name = "run-sklearn-transformer"
    outs = Param.odict([("x_processed", "x_test_processed",
                         "kale_dataset_id_local", "transformer_pipeline"],
                        step_name=name)

    def _submit_dataset_artifact(self, dataset):
        from kale.settings import settings
        from kale.marshal.utils import strip_marshal_path

        dataset_artifact = dataset.as_artifact()
        rok_version = self.vars.get("autosnapshot_start")
        if not dataset.artifact_uri and rok_version:
            dataset.artifact_uri = rokutils.get_uri_in_version(
                rok_version, strip_marshal_path(settings.INS["dataset"]))
```

```
            dataset_artifact = dataset.submit_artifact()
            return dataset_artifact

        def _link_input_artifacts(self, kale_dataset_id, kale_config_id):
            mlmd = mlmdutils.get_mlmd_instance()
            if kale_config_id > 0:
                mlmd.link_artifact_as_input(kale_config_id)
            if kale_dataset_id > 0:
                mlmd.link_artifact_as_input(kale_dataset_id)

        def _link_output_artifacts(self):
            from kale.common.artifacts import Transformer

            mlmd = mlmdutils.get_mlmd_instance()
            self.vars["transformer_artifact"] = Transformer(
                self.name, self.vars["preprocessor"]).submit_artifact()
            mlmd.link_artifact_as_output(self.vars["transformer_artifact"].id)

        def do_run(self, configuration, dataset, kale_dataset_id, kale_config_id,
                   masked_inputs):
            """Implementation of RunSKLearnTransformer."""
            import json
            import sklearn
            from kale.ml import utils

            kale_dataset_id = int(kale_dataset_id)
            kale_config_id = int(kale_config_id)
            if kale_dataset_id <= 0:
                kale_dataset_id = self._submit_dataset_artifact(dataset).id
            self._link_input_artifacts(kale_dataset_id, kale_config_id)

            configuration = utils.patch_configuration(configuration,
                                                      json.loads(masked_inputs),
                                                      coerce=True)
            steps = utils.get_sklearn_steps_list(configuration)
            self.vars["preprocessor"] = sklearn.pipeline.Pipeline(steps[:-1])
            x_processed = self.vars["preprocessor"].fit_transform(dataset.features,
                                                                  dataset.targets)
            x_test_processed = self.vars["preprocessor"].transform(
                dataset.features_test)

            self._link_output_artifacts()
            return (x_processed, x_test_processed, kale_dataset_id,
                    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 AutoMLCon guration Artifacts as inputs of this step.
   This step takes as inputs the IDs of the Kale Dataset and AutoMLCon guration Artifacts (pipeline parameters), so the linking is implemented via the _link_input_• artifacts method of the step.

2. Reads the input con guration . If the Con guration Run is part of a Katib experi• ment (created by the run•katib•experiment step) then we update the input con gu• ration, created by auto•sklearn, with the values that are suggested by Katib. In the case that the Con guration Run is created by the run•metalearning•con gurations step, we keep the con guration input as is. We implement this decision in the patch_con guration function of the kale.ml.utils module:

Listing 10.19: The patch_con guration function of the kale.ml.utils module

```
def patch _configuration (configuration : Configuration ,
                          overrides : Dict [str , Any],
                          coerce =False ):
"""Patch a configuration.

 Args:
      configuration: A suggested configuration extracted by auto-sklearn.
      overrides (dict): Override configuration values by providing them with
          keys matching the last token in the original Configuration keys.
          Provide an empty dictionary to make this function a no-op.
      coerce (bool): Convert the override value to the destination type.

 Returns:
      Configuration: Patched configuration.
 """
log .info ("Patching base configuration..." )
log .info ("Base %s" , configuration )

if not overrides :
    log .info ("No input parameters found to patch the base"
               " configuration." )
    return configuration

log .info ("Using the following configs to patch the base"
           " configuration: %s" , overrides )
patched _config = copy .deepcopy (configuration )
for name , override _value in overrides .items ():
    for key , conf _value in patched _config .get _dictionary ().items ():
        if key .split (":" )[•1] == name:
            if coerce :
                override _value = type (conf _value )(override _value )
            patched _config [key ] = override _value
log .info ("Patched %s" , patched _config )
return patched _config

```

3. Transforms the con guration 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

```
def get_sklearn_steps_list(config: Configuration):
    """Return a list of steps for sklearn Pipeline."""
    log.info("Getting SKLearn steps list...")
    if config.get("classifier:__choice__"):
        p = SimpleClassificationPipeline(config)
    elif config.get("regressor:__choice__"):
        p = SimpleRegressionPipeline(config)
    else:
        raise ValueError("No model found in input configuration")
    return p.steps
```

Essentially, this function initializes either a SimpleClassi cationPipeline 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 t_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 con guration, creates the model that the con guration describes and then trains the model on the processed training set that the previous step created. Let us view the class de nition and the main methods of the step:

Listing 10.21: The class de nition and the main methods of the train•sklearn•estimator step

```
class TrainSKLearnEstimator(PatchMaskedInputsMixin, Step):
    """Train a SKLearn estimator from an AutoML configuration.

    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 Con guration Run.
2. Reads the input marshalled con guration. If the Con guration Run is part of a Katib experiment (created by the run•katib•experiment step) then the do_run method updates the initial input con guration, created by auto•sklearn, with the values that are suggested by Katib. Otherwise, if the Con guration Run is created by the run•metalearning•con gurations step, we keep the input con guration as is. This decision is implemented in the patch_con guration function of the kale.ml.utils module. We exposed this function in listing 10.19.
3. Transforms the con guration 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 initial•izes either a SimpleClassi cationPipeline or SimpleRegressionPipeline from the au•tosklearn.pipeline (9.2.5) module and returns its steps list•attribute.
4. Keeps the last step of the pipeline con guration 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 con guration.
6. Trains the Pipeline object on the processed training dataset using its t meth•ods (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 InferSKLearn•Predictor 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 de nition and the main methods of the step:

Listing 10.22: The class de nition and the main methods of the infer•sklearn•predictor step

```
class InferSKLearnPredictor (Step ):
    """Run predictions on a trained model.

    Ins:
        model
        x_test _processed
        metric
        dataset
        task
        kale _model _id
        kale _dataset _id _local

    MLMD Inputs:
        kale.Model
        kale.Dataset

    MLMD Outputs:
        kale.TensorboardLogs
    """
    name = "infer-sklearn-predictor"
    has_metrics = True

    def _link _input _artifacts (self , kale _dataset _id , kale _model _id ):
        mlmd = mlmdutils .get _mlmd_instance ()
        mlmd.link _artifact _as_input (kale _dataset _id )
        mlmd.link _artifact _as_input (kale _model _id )

    def _update _model _artifact (self , kale _model _id , metric _name, metric _value ):
        mlmd = mlmdutils .get _mlmd_instance ()
        model = mlmdutils .get _artifact _by_id (kale _model _id )
        model _metrics = json .loads (model .properties ["metrics" ].string _value )
        model _metrics .update ({metric _name: metric _value })

        model _name = model .properties ["name" ].string _value
        class _name = model _name.split ("/" )[•1]
        new_name = "%s/%s" % (mlmd.name or "" , class _name)

        mlmdutils .patch _artifact _properties (kale _model _id ,
                                                {"name" : new_name,
                                                 "metrics" : model _metrics })

    def _produce _report (self , task , dataset , model ):
        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 Con guration 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 calcu• late_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 Con guration 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 di erent types of Artifacts that Kale creates in order to create a MLMD lineage throughout the AutoML experiment.

To be able to submit and update di erent types of MLMD Artifacts, we created the kale.common.artifacts module which contains class de nitions 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

```
class MLMDArtifact (abc.ABC):
    """A base class for objects submittable in MLMD,

    Other classes that correspond to MLMD Artifacts and that expect their instances to be
    submitted in MLMD should subclass this class.

    Example:

        >>> class MyClass(MLMDArtifact):
        ...
        ...     artifact_type_name = "kale.MyClass"
        ...     artifact_property_types = {"prop1": metadata_store_pb2.STRING,
        ...                                "prop2": metadata_store_pb2.INT}
        ...
        ...     @property
        ...     def artifact_properties(self):
        ...         prop1 = metadata_store_pb2.Value(
        ...             string_value=self.get_prop1())
        ...         prop2 = metadata_store_pb2.Value(
        ...             int_value=self.compute_prop2())
        ...         return {"prop1": prop1, "prop2": prop2}
        ...
        ....    @property
        ...     def artifact_custom_properties(self):
        ...         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.
             proto#L74-L81
34
35          Attributes:
36              artifact    _type _name (str): The name of the ArtifactType
37              artifact    _property   _types (dict): The mapping of property names and their
38                  MLMD types
39              artifact    _uri (str): The URI of the Artifact
40          """
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 AutoMLCon guration Artifact

One of our goals, regarding the MLMD lineage of the AutoML experiment, was to be able to retrieve information about the suggested con gurations that auto•sklearn provided. More speci cally, 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•con guration step of the AutoML Orchestrator is executed, it should create AutoMLCon guration 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 AutoMLCon guration Artifacts are outputs of the Execution that corresponds to the get•metalearning•con guration step.

These AutoMLCon guration Artifacts have a main property called model_data . Before the do_run method of the GetMetaLearningCon gurations step submits an AutoMLCon g• uration Artifact on MLMD, it lls this property (10.2.2.4) with a dictionary that contains:

1. The name of the model architecture of the suggested con guration.
2. The names and values of the parameters of that model.

The following snippet shows the class de nition, attributes and methods of the Au• toMLCon guration class.

Listing 10.24: The AutoMLCon guration class for creating and submitting Artifacts that correspond to pipeline con gurations

```
class AutoMLConfiguration (MLMDArtifact):
    """An AutoML configuration Artifact."""
    artifact_type_name = "kale.AutoMLConfiguration"
    artifact_property_types = {"model_data": metadata_store_pb2.STRING}

    def __init__(self, config_summary: Dict, run_id: str, estimator_name: str):
        self.config_summary = config_summary
        self.run_id = run_id
        self.estimator_name = estimator_name

    @property
    def artifact_properties(self) -> Dict[str, metadata_store_pb2.Value]:
        """Get kale.AutoMLConfiguration Artifact properties."""
        model_data = metadata_store_pb2.Value(
            string_value=json.dumps(self.config_summary))
        return {"model_data": model_data}

    @property
    def artifact_custom_properties(self):
        """Get kale.AutoMLConfiguration Artifact custom properties."""
        run_id_prop = metadata_store_pb2.Value(string_value=self.run_id)
        name_prop = metadata_store_pb2.Value(string_value=self.estimator_name)
        return {"run_id": run_id_prop,
                "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 speci c "entity" in the Kale•AutoML process that we built. Since they are not the main focus of the thesis, We will brie y 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 Con guration Run.
- Transformer : An Artifact to log the tted transformer that is produced during the run•sklearn•transformer step (10.3.2) of a Con guration Run.

10.4.2 The AutoMLExperiment 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 AutoMLExperiment object fetches and displays all the information that denotes the state of the AutoML process.

Let us provide a code snippet with the class de nition and the constructor method of the AutoMLExperiment object:

Listing 10.25: The AutoMLExperiment class

```
class AutoMLExperiment ():
    """A status tracker for the AutoML process."""
    def __init__(self, run_id: str):
        self.automl_orchestrator_run_id = run_id
        try:
            kfputils.get_run(run_id)
        except ApiException as e:
            if e.status == 404:
                raise ValueError("Invalid run ID: '%s'. The run ID you"
                                 " provided does not correspond to a KFP run."
                                 % run_id)
            raise RuntimeError("Failed to retrieve KFP run with ID '%s': %s"
                               % (run_id, str(e)))

        self.mlmd_client = mlmdutils.get_client()
        self.context_type_name = "KfpRun"
        self.context_name = mlmdutils.get_context_name_from_run_id(
            self.automl_orchestrator_run_id)

        self.context = None
        self.automl_configuration_artifacts = []
        self.automl_configuration_run_ids = []

        self._update_all_info()
```

The constructor method of the AutoMLExperiment initializes the object with just the run ID of the AutoML Orchestrator . Here is a list of the attributes of an AutoMLExperiment 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 AutoMLCon guration Artifacts
- the Con guration Run IDs

10.4.2.1 The list_con gurations Method

This API method shows information about the model architectures of the suggested con gurations. It manages to do that by calling the _update_all_info method that we described earlier, in order to collect all the AutoMLCon guration Artifacts (10.4.1.2) that were created by the get•metalearning•con gurations step (10.2.2).

Let us view the code of the list_con gurations method:

Listing 10.27: The list_con gurations method

```
def list_configurations(self):
    """Show information about the suggested model configurations."""
    self._update_all_info()
    if not self.automl_configuration_artifacts:
        print("There are no configurations yet...\n")
        return

    ml.utils.print_suggested_models(self.automl_configuration_artifacts)
```

As we can see from above, list_con gurations prints all the information related to the models of the pipeline con gurations 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

```
def print_suggested_models(artifacts: List[Artifact]):
    """Print suggested models given an MLMD artifacts list."""
    for idx, artifact in enumerate(artifacts, start=1):
        print("Configuration", idx)
        model_dict = json.loads(artifact.properties["model_data"].string_value)

        if "name" not in model_dict:
            raise RuntimeError("AutoMLConfiguration Artifact for"
                               " Configuration%d doesn't provide a name for"
                               " the estimator." % idx)
        print("Estimator:", model_dict["name"])

        if "parameters" not in model_dict:
            raise RuntimeError("AutoMLConfiguration Artifact for"
                               " Configuration%d doesn't provide any"
                               " parameters." % idx)
        print("Parameters:")

        for param in model_dict["parameters"]:
            print("    %s: %s" % (param, model_dict["parameters"][param]))
        print("\n")
```

The following image is an example of how the list_con gurations method outputs all the information regarding the models of the pipeline con gurations when ran in a Jupyter Notebook cell.

Figure 10.3: Example output of list_con gurations 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 Con guration Runs, as well as the metric scores that each Con guration Run produces.

To retrieve these information about the AutoML Orchestrator and the Con guration 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 Con guration Run IDs, summary calls the _update_all_info utility method of the AutoML• Experiment class.

Listing 10.29: The summary API method of the AutoMLExperiment class

```
def summary(self):
    """Show a summary of the AutoML process."""
    # Retrieve the current status of the AutoML Orchestrator.
    try:
        status = kfputils.get_run(
            self.automl_orchestrator_run_id).run.status
    except ApiException as e:
        if e.status == 404:
            raise ValueError("Invalid run ID: '%s'. The run ID you"
                             " provided does not correspond to a KFP run."
                             % self.automl_orchestrator_run_id)
        raise RuntimeError("Failed to retrieve KFP run with ID '%s': %s"
                           % (self.automl_orchestrator_run_id, str(e)))
    if not status:
        print("The AutoML orchestrator has not started yet...")
        return
    print("AutoML orchestrator status: %s\n" % status)

    self._update_all_info()
    if not self.automl_configuration_artifacts:
        print("There are no configurations yet...\n")
        return

    self._print_configuration_run_status_counts()
    if self.automl_configuration_run_ids:
        self._print_summary_table()
```

As it can be seen from the code snippet above, the summary method prints, serially, four di erent 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 Con guration Runs that have started.
3. A HTML table that counts the number of Con guration Runs that have a spe• ci c status. To implement this, summary uses another method of the AutoMLEx• periment class, called _print_con guration_run_status_counts . The following snippet shows the code of the _print_con guration_run_status_counts utility method:

Listing 10.30: The _print_con guration_run_status_counts method of the AutoMLExperi• ment class

```
    def _print_configuration_run_status_counts(self):
        """Print the status of each Configuration Run."""
        # Initially, print the number of started runs,
        # out of all the suggested Configurations.
        num_of_configs = len(self.automl_configuration_artifacts)
```

```
        num_of_config_runs = len(self.automl_configuration_run_ids)

        print("%d/%d Configuration runs have started.\n"
              % (num_of_config_runs, num_of_configs))

        # Then, print a table that shows how many runs
        # are currently "Running", "Failed", "Succeeded" etc...
        status_counts = {"Running": 0}
        for status in kfputils.KFP_RUN_FINAL_STATES:
            status_counts[status] = 0

        for run_id in self.automl_configuration_run_ids:
            try:
                status = kfputils.get_run(run_id).run.status
            except ApiException as e:
                if e.status == 404:
                    raise ValueError("Invalid run ID: '%s'."
                                     " The run ID you"
                                     " provided does not"
                                     " correspond to a KFP"
                                     " run." % run_id)
                raise RuntimeError("Failed to retrieve KFP run"
                                   " with ID '%s':"
                                   " %s" % (run_id, str(e)))

            if status not in status_counts:
                status_counts[status] = 0
            status_counts[status] += 1

        headers = ["Status", "Count"]

        if utils.is_ipython():
            # If we are running in a Jupyter Notebook we display
            # the table in HTML format.
            from IPython.core.display import display, HTML
            table = tabulate.tabulate(
                tabular_data=list(status_counts.items()),
                headers=headers,
                tablefmt="html")
            display(HTML(table))
        else:
            table = tabulate.tabulate(
                tabular_data=list(status_counts.items()),
                headers=headers)
            print(table)
        print("\n")
```

This method initially prints the number of started Con guration Runs out of all the

suggested pipeline con gurations. 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 Con gura• tion Run . To implement this, summary uses another method of the AutoMLExper• iment 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

```
    def _print_summary_table(self):
        """Print a summary table for the configuration runs."""
        tabular_data = []

        # This will be the header of the metrics column. When the first
        # (metric_name, score) pair is received for a configuration run,
        # the header will be changed to "Metric (<metric_name>)".
        metric_header = "Metric"

        for i, run_id in enumerate(self.automl_configuration_run_ids):
            # Get the status of the configuration run that has started.
            try:
                status = kfputils.get_run(run_id).run.status
            except ApiException as e:
                if e.status == 404:
                    raise ValueError("Invalid run ID: '%s'."
                                     " The run ID you"
                                     " provided does not"
                                     " correspond to a KFP"
                                     " run." % run_id)
                raise RuntimeError("Failed to retrieve KFP run"
                                   " with ID '%s':"
                                   " %s" % (run_id, str(e)))
            metric = "-"

            # Get the MLMD Context of the configuration run, if exists.
            context_name = mlmdutils.get_context_name_from_run_id(run_id)
            context = self.mlmd_client.get_context_by_type_and_name(
                type_name=mlmdutils.RUN_CONTEXT_TYPE_NAME,
                context_name=context_name)

            if not context:
                tabular_data.append([i + 1, run_id, status, metric])
                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.

Figure 10.4: Example output of summary method when executed in a Jupyter Notebook cell

Figure 10.5: The KFP UI page for a Con guration 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 eld of AutoML experiment libraries. More speci cally, 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 classi cation 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 speci cally, 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 su cient.

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 ve pipeline con gurations (number_of_con gurations=5) while at most two Con guration Runs could run in parallel (max_parallel_con gurations=2). Similarly, the Katib experiment that implements the hyper•parameter optimization part of our Kale•AutoML process ran ve 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

Classi er and AutoSklearnRegressor API objects. The input parameter con gurations 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 AutoSklearnClassi er 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	0.95
CIFAR-10	accuracy	0.25	0.31	0.25
GERMAN CREDIT DATA	accuracy	0.776	0.78	0.78
WINE QUALITY	mean squared error	0.506	0.478	0.457
DIABETES	mean squared error	42.962	42.539	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 con• struction mechanism demonstrates a signi cant improvement in test scores in compari• son 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 ve con gu• ration 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 Con guration Runs was not su cient 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 nally wrap this thesis up with the future research directions of our mecha• nism. We plan on pursuing these actively over the next months or years.

- Extend our AutoMLExperiment (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 Con guration 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 Con guration 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 kai Frank Hutter. E cient and Robust Automated Machine Learning . Advances in Neural Information Processing Systems 28 C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama kai R. Garnett, epimelhtèc , selÐdec 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 kai E. Duchesnay. Scikit•learn: Machine Learning in Python . Journal of Machine Learning Research , 12:2825 2830, 2011.

[3] The o cial Kube ow page . https://www.kubeflow.org/ . Accessed: 31•10•2021.

[4] Johnu George, Ce Gao, Richard Liu, Hou Gang Liu, Yuan Tang, Ramdoot Pydipaty kai Amit Kumar Saha. A Scalable and Cloud•Native Hyperparameter Tuning System , 2020.

[5] Kube ow'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) . Accessed: 01•11•2021.

[7] Kubernetes' o cial web•page . https://kubernetes.io/ . Accessed: 30•10•2021.

[8] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li kai 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, Sa a Abdalla kai Carol Willing. Jupyter Notebooks a publishing format for reproducible computational work ows . Positioning and Power in Academic Publishing: Players, Agents and Agendas F. Loizides kai B. Schmidt, epimelhtèc , selÐdec 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 o□cial 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 Fernándezdel 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, Antônio 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 o□cial etcd web-page. https://etcd.io/. Accessed: 30-10-2021.

[20] The o□cial TensorFlow page. https://www.tensorflow.org/. Accessed: 31-10-2021.

[21] The o□cial TensorFlow Extended (TFX) page. https://www.tensorflow.org/tfx/. Accessed: 31-10-2021.

[22] The o□cial website of Argo Project. https://argoproj.github.io/l. 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 o□cial 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-a4ede38bea1fl. 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 Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney και 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). 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.