National Technical University of Athens
School of Electrical and Computer Engineering
MSc Data Science and Machine Learning

Hybrid Recommendation Systems using Neural Networks

Diploma Thesis

of

MICHAIL V. BIZIMIS

Supervisor: Alexandros Potamianos
Professor NTUA

Athens, October 2022

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
MSc DATA SCIENCE AND MACHINE LEARNING

Hybrid Recommendation Systems using Neural Networks

DIPLOMA THESIS

of

MICHAIL V. BIZIMIS

Supervisor: Alexandros Potamianos
Professor NTUA

Approved by the examination committee on the 27th of October 2022.

(Signature)	*(Signature)*	*(Signature)*
....	....	....
Alexandros Potamianos	Theodoros Giannakopoulos	Symeon Papavassiliou
Professor NTUA	Researcher NCSR Demokritos	Professor NTUA

Athens, October 2022

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

....
Michail V. Bizimis
27th October 2022

Abstract

As consumers of the 21st century, deciding where to focus our limited time and attention, when faced with making a choice over a vast number of alternatives, can quickly become overwhelming. To alleviate this problem, recommendation systems, capable of automatically filtering these options and suggesting to us only a smaller subset of relevant ones, tailored to our preferences, have been widely employed. In this thesis, we develop a hybrid recommendation system, based on three increasingly complex neural network architectures, which we then apply for movie recommendations.

More specifically, we start by combining the Neural Collaborative Filtering framework, a Collaborative Filtering method based on neural networks, with content-based methods for creating item and user profiles, in order to acquire said hybrid recommendation system. Then, we extend this architecture in two separate ways. In the first, instead of relying on fixed user profiles, we create them dynamically during the forward pass, wherein each aggregated item profile that is part of the user profile is assigned a different weight as a result of an item-item attention mechanism. This mechanism, not only improves the model's performance, but also offers some useful explainability. In the second, we revert back to using fixed user profiles, except now we incorporate Graph Neural Networks into the embedding process, in an attempt to explicitly capture the collaborative signal on the user-item bipartite graph and, in this way, acquire better user and item embeddings.

For the purposes of this thesis, we create a custom dataset by combining of a popular Collaborative Filtering dataset for movies with movie metadata as their content. We proceed to train models from the three aforementioned architectures on it, under both a regression and a ranking setting and, then, we evaluate and compare them using suitable regression as well as ranking metrics. In our experiments, we found that the regression training setting works best. Out of the three architectures, the second performs the best on the test set, whilst simultaneously offering some much desired explainability. On the contrary, we found that the third did not perform any better on our dataset than the much simpler first one, possibly due to the lack of complicated enough patterns in it. Finally, to showcase our best model, we deploy it in a demo web application, where a user can receive explainable movie recommendations after rating movies in our dataset.

Keywords

Recommendation Systems, Hybrid Recommendation Systems, Deep Learning, Neural Networks, Collaborative Filtering, Content-based profiles, Neural Collaborative Filtering, Attention, Graph Neural Networks, Neural Graph Collaborative Filtering

Περίληψη

Ως καταναλωτές του 21ου αιώνα, είναι δύσκολο να επιλέξουμε που να εστιάσουμε την προσοχή και τον χρόνο μας, όταν καλούμαστε να πάρουμε μια απόφαση ανάμεσα σε ένα τεράστιο πλήθος επιλογών. Για την ελάττωση αυτού του προβλήματος, συστήματα συστάσεων, ικανά να εξετάζουν αυτόματα αυτές τις επιλογές και να μας προτείνουν μόνο ένα μικρότερο υποσύνολο τους, έχουν χρησιμοποιηθεί ευρέως. Σε αυτήν τη διπλωματική εργασία, αναπτύσσουμε ένα υβριδικό σύστημα συστάσεων, βασισμένο σε τρεις αύξουσας πολυπλοκότητας αρχιτεκτονικές νευρωνικών δικτύων, το όποιο εφαρμόζουμε για τη σύσταση ταινιών.

Πιο συγκεκριμένα, ξεκινάμε συνδυάζοντας μία μέθοδο Συνεργατικού Φιλτραρίσματος, που βασίζεται σε νευρωνικά δίκτυα, με μεθόδους δημιουργίας προφίλ χρηστών και αντικειμένων βασισμένα στο περιεχόμενο, έτσι ώστε να καταλήξουμε με ένα υβριδικό σύστημα συστάσεων βασισμένο σε νευρωνικά δίκτυα. Ύστερα, επεκτείνουμε αυτήν την αρχιτεκτονική με δύο διαφορετικούς τρόπους. Στον πρώτο, αντί να στηριχθούμε σε στατικά προφίλ χρηστών, τα δημιουργούμε δυναμικά, ως μέρος του νευρωνικού δικτύου, υπολογίζοντας το βάρος με το οποίο θα συναθροιστεί κάθε αντικείμενο-μέρος του προφίλ μέσω ενός μηχανισμού προσοχής μεταξύ δύο αντικειμένων. Αυτός ο μηχανισμός προσφέρει, όχι μόνο καλύτερη απόδοση, αλλά και έναν βαθμό επεξηγησιμότητας. Στον δεύτερο, επιστρέφουμε στη χρήση στατικών προφίλ χρήστη, αλλά τώρα ενσωματώνουμε Νευρωνικά Δίκτυα Γράφων στη διαδικασία παραγωγής διανυσματικών αναπαραστάσεων για χρήστες και αντικείμενα, έτσι ώστε να συλλέξουμε άμεσα το συνεργατικό σήμα του διμερούς γράφου χρηστών-αντικειμένων.

Στα πλαίσια αυτής της εργασίας, δημιουργούμε ένα δικό μας σύνολο δεδομένων, συνδυάζοντας ένα δημοφιλές σύνολο δεδομένων Συνεργατικού Φιλτραρίσματος για ταινίες με μεταδεδομένα ταινιών. Στη συνέχεια, εκπαιδεύουμε μοντέλα από τις τρεις προαναφερθείσες αρχιτεκτονικές σε αυτό για παλινδρόμηση αλλά και για κατάταξη και, έπειτα, τα αξιολογούμε με αντίστοιχες μετρικές. Στα πειράματά μας, διαπιστώσαμε ότι η εκπαίδευση για παλινδρόμηση είναι προτιμότερη, ότι η δεύτερη αρχιτεκτονική αποδίδει καλύτερα, ενώ ταυτόχρονα προσφέρει έναν σημαντικό βαθμό επεξηγησιμότητας, και ότι η τρίτη αρχιτεκτονική δεν αποδίδει καλύτερα από την απλούστερη πρώτη στα δικά μας δεδομένα. Τέλος, δημιουργούμε μία ενδεικτική εφαρμογή διαδικτύου, όπου ένας χρήστης μπορεί να λάβει επεξηγήσιμες συστάσεις ταινιών, έχοντας πρώτα βαθμολογήσει ταινίες του συνόλου δεδομένων μας.

Λέξεις Κλειδιά

Υβριδικά Συστήματα Συστάσεων, Βαθιά Μάθηση, Νευρωνικά Δίκτυα, Συνεργατικό Φιλτράρισμα, Προφίλ βασισμένα στο περιεχόμενο, Προσοχή, Νευρωνικά Δίκτυα Γράφων

Σύνοψη

Στη σημερινή εποχή, είναι αναμφισβήτητο ότι απολαμβάνουμε ένα υψηλότερο βιοτικό επίπεδο σε σύγκριση με άλλες εποχές της ιστορίας μας ως άνθρωποι. Χάρις στην τεχνολογική και βιομηχανική μας πρόοδο, ποτέ πριν δεν είχαμε τη δυνατότητα να έχουμε τόσο εύκολη πρόσβαση σε τέτοια αφθονία από καταναλωτικά αγαθά και υπηρεσίες. Ωστόσο, με τόσες πολλές εναλλακτικές, βρισκόμαστε συνεχώς αντιμέτωποι με έναν συντριπτικό αριθμό από διαφορετικές επιλογές, από τα άρθρα ειδήσεων που διαβάζουμε μέχρι τις ταινίες που παρακολουθούμε. Σε πολλές περιπτώσεις, το να εξετάσουμε όλες τις διαθέσιμες επιλογές, προκειμένου να πάρουμε μία ενημερωμένη απόφαση, μπορεί να είναι δύσκολο, χρονοβόρο και κουραστικό, οδηγώντας σε ένα φαινόμενο γνωστό ως *υπερφόρτωση επιλογών*[1]. Το φαινόμενο αυτό περιγράφει μια γνωστική κατάσταση, κατά την οποία οι άνθρωποι δυσκολεύονται να λάβουν μία απόφαση, όταν έρχονται αντιμέτωποι με πολλές επιλογές. Για την ελάττωση αυτού του προβλήματος, *συστήματα συστάσεων* έχουν αναπτυχθεί και χρησιμοποιηθεί ευρέως. Αυτά τα συστήματα έχουν τη δυνατότητα να ξεσκαρτάρουν τις άσχετες επιλογές και να προτείνουν στους χρήστες τους μόνο ένα μικρότερο σύνολο από τις πιο σχετικές επιλογές για αυτούς, κάνοντας, έτσι, την τελική επιλογή των χρηστών τους πολύ πιο εύκολη.

Υπάρχουν πολλοί τύποι συστημάτων συστάσεων, αλλά γενικά οι δύο κύριες κατηγορίες που έχουν καθιερωθεί είναι οι μέθοδοι Συνεργατικού Φιλτραρίσματος και οι μέθοδοι βασισμένες στο περιεχόμενο. Και τα δύο αυτά σύνολα μεθόδων βασίζονται σε παρελθοντικές αλληλεπιδράσεις ενός χρήστη με αντικείμενα, προκειμένου να του κάνουν συστάσεις για νέα αντικείμενα, που είναι εξατομικευμένα σε αυτόν. Ωστόσο, οι πρώτες το πετυχαίνουν αυτό αξιοποιώντας συσχετίσεις μεταξύ των αλληλεπιδράσεων του χρήστη με άλλες γνωστές αλληλεπιδράσεις χρήστη-αντικειμένου από άλλους παρόμοιους χρήστες, ενώ οι τελευταίες το πετυχαίνουν εστιάζοντας στο περιεχόμενο των αντικειμένων που ο εν λόγω χρήστης έχει αλληλεπιδράσει στο παρελθόν. Και οι δύο αυτές κατηγορίες έχουν τα θετικά και τα αρνητικά τους. Η επιτυχία των μεθόδων βασισμένων στο περιεχόμενο εξαρτάται από την ποιότητα των χαρακτηριστικών των αντικειμένων. Οι μέθοδοι Συνεργατικού Φιλτραρίσματος, από την άλλη, δεν απαιτούν πρόσβαση σε τέτοια χαρακτηριστικά, αλλά συνήθως δεν μπορούν να αντιμετωπίσουν νέους χρήστες και αντικείμενα, δεν τα πηγαίνουν πολύ καλά, όταν δεν υπάρχουν αρκετές γνωστές αλληλεπιδράσεις μεταξύ χρηστών και αντικειμένων, και είναι πιο δύσκολο να κλιμακωθούν σε πολύ μεγάλο αριθμό από χρήστες και αντικείμενα. Για να πάρουμε τα καλύτερα και από τους δύο κόσμους ή να αντιμετωπίσουμε τα μειονεκτήματα του ενός, καθώς και να πετύχουμε καλύτερη επίδοση συνολικά, υβριδικά συστήματα συστάσεων συνδυάζουν μεθόδους και από τις δύο αυτές κατηγορίες.

[1] `https://en.wikipedia.org/wiki/Overchoice`

Την τελευταία δεκαετία, τα βαθιά νευρωνικά δίκτυα έχουν επιτύχει κορυφαίες επιδόσεις σε πολλές εφαρμογές. Υπό το φως αυτής της ραγδαίας εξέλιξης της βαθιάς μάθησης, πολλά συστήματα συστάσεων, που βασίζονται σε μοντέλα μηχανικής μάθησης, έχουν επανασχεδιαστεί έτσι, ώστε να ενσωματώνουν βαθιά νευρωνικά δίκτυα, προκειμένου να βελτιώσουν περαιτέρω την επίδοσή τους. Το Neural Collaborative Filtering [11] αποτελεί μία τέτοια επέκταση μίας δημοφιλής μεθόδου Συνεργατικού Φιλτραρίσματος γνωστή ως Matrix Factorization[2], στην οποία ένα νευρωνικό δίκτυο χρησιμοποιείται για να μοντελοποιήσουμε μία συνάρτηση που αντιστοιχεί ζευγάρια από χρήστες και αντικείμενα σε μία τιμή προτίμησης (πχ μια βαθμολογία). Αυτή η μέθοδος παρουσιάστηκε αρχικά ως μία μέθοδος αμιγούς Συνεργατικού Φιλτραρίσματος, αφού χρησιμοποιεί one-hot διανύσματα για να αναπαραστήσει χρήστες και αντικείμενα. Παρόλα αυτά, οι συγγραφείς αναφέρουν τη δυνατότητα χρήσης διαφορετικών διανυσμάτων για αυτό τον σκοπό.

Αυτό που προτείνουμε εμείς είναι ένα υβριδικό σύστημα συστάσεων, βασισμένο στο Neural Collaborative Filtering, αλλά στο οποίο αναπαριστούμε χρήστες και αντικείμενα με προφίλ που βασίζονται στο περιεχόμενο, αντί για one-hot διανύσματα. Δηλαδή αναπαριστούμε κάθε αντικείμενο με ένα διάνυσμα που αντιπροσωπεύει τα χαρακτηριστικά του και κάθε χρήστη με ένα διάνυσμα που δημιουργείται από την κατάλληλη συνάθροιση των διανυσμάτων χαρακτηριστικών των αντικειμένων με τα οποία έχει αλληλεπιδράσει στο παρελθόν. Με αυτόν τον τρόπο, περιμένουμε ότι χρήστες με παρόμοιες προτιμήσεις στα χαρακτηριστικά των αντικειμένων θα καταλήξουν να έχουν πιο παρόμοια προφίλ σε σχέση με χρήστες με διαφορετικές προτιμήσεις. Χάρις σε αυτή τη χρήση προφίλ βασισμένα στο περιεχόμενο, το μοντέλο μας μπορεί να γενικεύει σε νέα αντικείμενα και, το σημαντικότερο, σε νέους χρήστες, χωρίς να χρειάζεται να το επανεκπαιδεύσουμε, λύνοντας έτσι το cold-start πρόβλημα, το οποίο ταλανίζει τις μεθόδους Συνεργατικού Φιλτραρίσματος.

Πιο συγκεκριμένα, αναπτύσσουμε τρεις διαφορετικές αρχιτεκτονικές νευρωνικών δικτύων. Η πρώτη, στην οποία θα αναφερόμαστε ως Basic NCF, χρησιμοποιεί στατικά προφίλ χρηστών και αντικειμένων ως είσοδο στο νευρωνικό δίκτυο. Η δεύτερη, που ονομάζουμε Attention NCF, δημιουργεί τα προφίλ των χρηστών δυναμικά, κατά το εμπρόσθιο πέρασμα του δικτύου, όπου, εμπνευσμένα από τη δουλειά στο [28], υπολογίζει ένα βάρος προσοχής μεταξύ του αντικειμένου της εισόδου και κάθε αντικειμένου που θα χρησιμοποιηθεί ως μέρος του προφίλ του χρήστη. Χρησιμοποιώντας αυτόν τον μηχανισμό προσοχής, μπορεί να καταλήξει σε διαφορετικό προφίλ για τον ίδιο χρήστη, ανάλογα με το αντικείμενο της εισόδου. Όπως θα δούμε, αυτό, όχι μόνο οδηγεί σε καλύτερη επίδοση, αλλά προσφέρει και έναν σημαντικό βαθμό επεξηγησιμότητας, καθώς μπορούμε να προσφέρουμε αντικείμενα με μεγάλη προσοχή ως μία εξήγηση του γιατί ένα αντικείμενο συστάθηκε στον χρήστη. Τέλος, η τρίτη, που ονομάζουμε Graph NCF, χρησιμοποιεί πάλι στατικά προφίλ χρηστών και αντικειμένων, όπως η πρώτη, αλλά προσπαθεί να μάθει καλύτερες διανυσματικές αναπαραστάσεις για τους χρήστες και τα αντικείμενα, εκμεταλλευόμενη με άμεσο τρόπο το συνεργατικό σήμα στον διμερή γράφο χρηστών-αντικειμένων, χρησιμοποιώντας νευρωνικά δίκτυα γράφων, ακολουθώντας τις δουλειές στα [29] και [10].

[2]https://en.wikipedia.org/wiki/Matrix_factorization_(recommender_systems)

Συνολικά, εκτός από την ανάπτυξη των τριών παραπάνω αρχιτεκτονικών νευρωνικών δικτύων, θα λέγαμε ότι η συνεισφορά αυτής της διπλωματικής εργασίας περιλαμβάνει τα εξής:

- Δημιουργούμε ένα νέο σύνολο δεδομένων, συνδυάζοντας ένα δημοφιλές σύνολο δεδομένων Συνεργατικού Φιλτραρίσματος με βαθμολογίες χρηστών σε ταινίες μαζί με μεταδεδομένα για αυτές (πχ είδος, ηθοποιοί, κτλ.) ως τα χαρακτηριστικά τους.

- Αφού χωρίσουμε κατάλληλα τις αλληλεπιδράσεις (βαθμολογίες) του συνόλου δεδομένων μας σε σύνολα εκπαίδευσης, επικύρωσης και ελέγχου, εκπαιδεύουμε μοντέλα από τις τρεις προαναφερθείσες αρχιτεκτονικές νευρωνικών δικτύων σε αυτές, τα οποία αξιολογούμε και συγκρίνουμε με μετρικές παλινδρόμησης και μετρικές κατάταξης.

- Για την εκπαίδευση τους, εξερευνούμε δύο δημοφιλείς προσεγγίσεις: την εκπαίδευσή τους για παλινδρόμηση με τη συνάρτηση κόστους MSE και την εκπαίδευσή τους για κατάταξη με τη συνάρτηση κόστους BPR.

- Δεδομένου ότι το Attention NCF προσφέρει την προαναφερθείσα επεξηγησιμότητα, απεικονίζουμε διαισθητικά τον μηχανισμό προσοχής του, υπολογίζοντας κάποια στατιστικά για τα βάρη προσοχής καθώς προβλέπουμε το σύνολο ελέγχου.

- Δημιουργούμε μία ενδεικτική εφαρμογή διαδικτύου, στην οποία εκθέτουμε το καλύτερο μοντέλο από τα πειράματά μας, που κατέληξε να είναι το Attention NCF. Σε αυτήν, ένας χρήστης μπορεί να βαθμολογήσει ταινίες και να λάβει επεξηγήσιμες συστάσεις, στις οποίες φαίνονται τα ήδη βαθμολογημένα αντικείμενα τα οποία πήραν το μεγαλύτερο βάρος προσοχής.

Όλος ο κώδικας αυτής της διπλωματικής εργασίας, συμπεριλαμβανομένου της ενδεικτικής εφαρμογής διαδικτύου, είναι open-source και είναι διαθέσιμος στο εξής δημόσιο git repository: `https://github.com/michaelbzms/DeepRecommendation`.

"Smart people focus on the right things."

- Jensen Huang

Acknowledgements

First of all, I would like to thank my professor, Dr. Alexandros Potamianos, and the researcher Dr. Theodoros Giannakopoulos, for giving me the opportunity to undertake this thesis, concerning the application of neural networks on a subject that I find very interesting. Furthermore, I would like to thank Mr. Giannakopoulos and, especially, the assistant researcher Konstantinos Bougiatiotis, for their constant support and guidance throughout the implementation and writing of this thesis. Finally, I would like to thank my family and friends who supported me during this period.

Athens, October 2022

Michail V. Bizimis

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In this day and age, it is no secret that we enjoy a higher standard of living compared to other eras of our history as humans. Thanks to our technological and industrial prowess, never before have we grown so accustomed to such an abundance of consumer goods and services. With so many alternatives to choose from, however, we are constantly faced with an overwhelming amount of options, from the news articles we read to the movies we watch. In many cases, going through all the available options, in order to make an informed decision, can be intimidating, time-consuming and exhausting, leading to a phenomenon known as *choice overload* or *overchoice*[1]. Overchoice is a cognitive impairment in which people have a difficult time making a decision when faced with many options. To amend this issue, *recommendation systems* have emerged. These systems are able to discard irrelevant options and only present their users with a smaller and easier to manage subset of relevant ones, usually in the form of top-K recommendations, thereby making their user's choice a much easier task.

There exist many types of recommendation systems, but the two main distinct categories have generally been Collaborative Filtering methods and content-based methods. Both of these sets of methods rely on a user's past interactions with items, in order to make new item recommendations that are personalized to him. However, the former do so by leveraging correlations between the user's own past interactions and other known user-item interactions from other similar users, while the latter do so by focusing on the content of the items that the user in question has interacted with in the past. Each of these two categories comes with its own advantages and disadvantages. Content-based methods rely on the quality of item features to be successful. Collaborative Filtering methods, on the other hand, do not require access to such features, but they usually cannot deal with new users and items and they have been known to struggle when there are a lot of missing interactions as well as with scaling to a significantly large number of users and/or items. To get the best of both worlds or to combat one's drawbacks, as well as achieve better performance overall, hybrid recommendation systems combine methods from both of these categories.

[1] https://en.wikipedia.org/wiki/Overchoice

Over the last decade, deep neural networks have achieved state-of-the-art performance in many applications. In light of this rise of deep learning, many model-based recommendation systems have been redesigned to incorporate deep neural networks in order to improve their performance. Neural Collaborative Filtering [11] is one such extension of a popular model-based Collaborative Filtering method known as Matrix Factorization[2], wherein a neural network is used in order to model a function that maps user-item pairs to a preference score (e.g. a rating). This method was originally presented as a purely Collaborative Filtering one, seeing as it uses one-hot vectors to represent users and items. Nevertheless, the authors do mention the possibility of using other meaningful vector representations for items and/or users.

What we propose is a hybrid recommendation system, based on Neural Collaborative Filtering, but wherein we represent users and items with content-based profiles instead of one-hot vectors. That is, we represent an item with a feature vector and a user with a vector that is created by appropriately aggregating the feature vectors of items that the user has interacted with in the past. Via this aggregation, we are trying to estimate the user's preferences in items with respect to their features. Hence, we expect users with similar preferences to have more similar profiles than users with different ones. It is this use of content-based profiles that allows our model to generalize to new items and, more importantly, new users, without having to retrain, thereby solving the cold-start problem associated with Collaborative Filtering methods.

More specifically, we develop three such distinct neural network architectures. The first, which we shall refer to as Basic NCF, uses fixed user and item profiles as input vectors to the neural network. The second, Attention NCF, creates the user profiles dynamically, during the forward pass, where, inspired from the work in [28], it calculates an attention weight between the input item and every interacted item that will make up the user profile. Using this item-item attention mechanism, it has the ability to end up with a different profile for the same user, depending on the input item, for which it is being called to predict a preference score. As we shall see, this does not only result in better performance, but also in some much needed explainability, as we can offer highly attended interacted items as an explanation for why an item is being recommended to a user. Last but not least, the third architecture, Graph NCF, uses fixed user and item profiles, like the first one, but it also attempts to learn better user and item embeddings by explicitly capturing the collaborative signal on the user-item bipartite graph via Graph Neural Networks, following the work in [29] and [10].

[2] https://en.wikipedia.org/wiki/Matrix_factorization_(recommender_systems)

1.2 Thesis Contribution

All in all, other than the development of these three neural network architectures, this thesis's contribution can be summarized as follows:

- We create a custom dataset for hybrid recommendation by combining a popular Collaborative Filtering dataset of user ratings to movies with movie metadata (e.g. genres, actors, etc.) as features for those movies.
- After appropriately splitting the dataset's user-item interactions (ratings) into train, validation and test sets, we train models from the three aforementioned neural network architectures on it and we evaluate and compare them using both regression and ranking metrics.
- For training, we explore two popular formulations, training them for regression using the MSE loss and training them for ranking using the BPR loss.
- Considering that Attention NCF offers the previously mentioned explainability, we intuitively visualize its attention mechanism via some aggregated statistics on the attention weights produced while running inference on the test set.
- We create a demo web application where we showcase the best model from our experiments, which turned out to be Attention NCF. In this application, a user can rate movies and receive explainable recommendations for new ones, where highly attended rated movies will be shown for each recommendation.

All the source code for this thesis, including the web application, can be found at this public git repository: https://github.com/michaelbzms/DeepRecommendation.

1.3 Thesis organization

The rest of this thesis can be broken down as follows:

- In Section 2, we delve into some important background for recommendation systems in general as well as related work, which also employs neural networks for recommendations.
- In Section 3, we discuss our approach to creating content-based profiles for users and items and, then, we present in detail the three aforementioned architectures: Basic NCF, Attention NCF and Graph NCF. We also look into the two different formulations that we used for training these models.
- In Section 4, we describe how we create and split the custom dataset that we use. We go into the evaluation metrics we employed and present the most interesting experiments we conducted along with their results.
- In Section 5, we review our work and report our conclusions along with possible directions for future work.

Chapter 2

Background and related work

2.1 Introduction to recommendation systems

2.1.1 The recommendation setting

In the recommendation setting, we usually assume that we have access to certain users, certain items and recorded interactions between them. In this context, the general goal of a recommendation system is to learn to predict potential future interactions between users and items from those recorded interactions and/or additional auxiliary features for the items and/or the users.

We typically store recorded interactions between users and items in a matrix dubbed as the *utility matrix*. This matrix is usually a two-dimensional matrix with one dimension for all the users and one dimension for all the items. Each cell represents a user-item interaction and its value may be known (for recorded interactions) or unknown. As there usually does not exist a recorded interaction between every user and every item, this matrix is usually quite sparse.

The type of values in the utility matrix depend on the type of interactions we have access to. We may have access to:

1. Explicit interactions, where the user has explicitly stated his preference or dispreference for an item. These usually come in the form of *user ratings*, e.g. discrete integer values on a scale 1-5 (5-star rating system), in the range {-1, 0, 1} (like/dislike), continuous values in [0, 1], etc.

2. Implicit interactions, where we derive user preferences implicitly from their activity (e.g. their views, their purchases, etc.). In these situations, we usually only know that there exists a positive (i.e. unary data) or positive/negative (i.e. binary data) interaction between a user-item pair.

Explicit interactions

Explicit interactions are of course more informative, since we can convert them to binary or unary implicit ones by applying a threshold (e.g. if the rating is $\geq$ than the average rating). At the same time, however, it is rarer to have access to explicit interactions, since users may not easily volunteer feedback that requires effort on their part such as rating an item, and, even if we do have access to some, they may be too few to work with.

	item 1	item 2	item 3	item 4
user 1	2	5	1	3
user 2	4	?	?	1
user 3	?	4	2	?
user 4	2	4	3	1
user 5	1	3	2	?

Figure 2.1. *A simple example for a utility matrix with explicit ratings.*

Implicit interactions

On the other hand, implicit interactions such as users clicking on an article, purchasing an item, watching a video, etc. are much easier to collect in large quantities without requiring any effort from the user. Thus, they tend to be more common.

At the same time, however, they are trickier to work with because, unless we also have access to negative interactions (e.g. a user refunding an item), there is no way to tell for sure if the lack of an interaction between a user and an item is because the user is not interested in the item or because he has not yet been presented with it.

In other words, when working with implicit recorded interactions that are *only positive* (i.e. unary data), we have to make an assumption, which may not always be true, about which blank entries in the utility matrix are to be considered negative interactions. Whereas when working with explicit recorded interactions we typically won't have to use blank entries at all, since the known interactions can be both positive and negative, as quantified by a scalar value like a rating.

We have to make this assumption because we typically need both positive and negative interactions to train a discriminative machine learning model to perform recommendations. This type of learning, where we only have positive labeled samples and the unlabeled samples could be both positive and negative, is usually referred to as *PU learning* (Positive and Unlabeled) [2]. In the context of recommendation, it is also referred to as *One-Class Collaborative Filtering* [21].

	item 1	item 2	item 3	item 4
user 1	?	1	?	1
user 2	1	?	?	?
user 3	?	1	?	?
user 4	?	1	1	?
user 5	?	1	?	?

(a) *Without negative interactions (unary data).*

	item 1	item 2	item 3	item 4
user 1	0	1	0	1
user 2	1	?	?	0
user 3	?	1	0	?
user 4	0	1	1	0
user 5	0	1	0	?

(b) *With negative interactions (binary data).*

Figure 2.2. *The same example for a utility matrix, but with implicit interactions. On the left (a) we can see unary data, while on the right (b) we can see binary data.*

2.1.2 Defining the task of recommendation

When defining the task of recommendation, there are two primary formulations to consider [1]:

1. In the *prediction version* of the problem, we are given a user u and an item i and we want to learn to predict a *preference score* $f(u, i)$ between them. This preference score, depending on whether we are working with explicit or implicit interactions, could be a *rating* from the user to the item (i.e. *regression*) or the *probability* of a future interaction happening (i.e. *binary classification*) respectively. Under this formulation, we are essentially trying to fill in the blanks of the utility matrix.
2. In the *ranking version* of the problem on the other hand, we do not have to predict a preference score, we only care about learning to make *top-k recommendations* of items for each user (or, more rarely, top-k recommendations of users for each item). In this scenario, we typically want to *learn to rank* [15] all the items per user.

Of course, the prediction version is more general as if we solve the prediction problem we already have a solution for the ranking problem: just sort the items in descending preference score for a user and recommend the top-k ones. In some cases, however, it may be more natural to solve the ranking version (e.g. in purely content-based methods) and, if we only care about making top-k item recommendations to users, then it may as well be all we need.

2.2 Traditional approaches

There traditionally have been three main approaches to solving the recommendation problem: Collaborative Filtering methods, content-based methods and hybrid methods, i.e. the combination of the two.

2.2.1 Collaborative Filtering methods

Collaborative Filtering (CF) methods typically solve the prediction version of the recommendation problem. They predict the missing values in the utility matrix by leveraging the information already on it. That is, they rely on recorded past interactions between users and items to predict future ones. They are called "collaborative" because they attempt to capture similarities/correlations between users and/or between items in order to make recommendations. For example, recommend to a user u an item i that other similar-to-him users liked.

Collaborative Filtering methods can be further split into memory-based methods and model-based methods [1].

Memory-based methods

Memory-based methods, aka *neighborhood-based collaborative filtering* [1], are simple heuristic similarity-based methods. The main idea is to represent users with their rows and items with their columns in the utility matrix. For example, in Figure 2.1, user 5 can be represented by the vector (with missing values) $[1, 3, 2, \_]$ while item 1 is represented by the vector $[2, 4, \_, 2, 1]$.

Then, using user-based CF, to predict the value $f(u, i)$ of the cell for user u and item i we perform the two following steps [14]:

1. First, find the k users (rows) most similar to u, $n_i \in \mathit{NearestNeighbors}(u)$ $i = 1 \ldots k$, via some vector similarity metric (e.g. cosine similarity, Jaccard similarity, etc.), that can deal with missing values (e.g. treat them as zeroes).

2. Then, aggregate (e.g. average) the *known* interactions $f(v_i, i)$ between those k neighboring users and item i to predict $f(u, i)$.

After performing the first step and finding user u's k-nearest neighbors, we can fill all the blanks in the row for user u. Consequently, we can immediately make recommendations for user u.

Due to symmetry, we can also do the same thing from the item's perspective, aka item-based CF. Item-based CF consists of finding the k most similar items to i and then aggregating the *known* preference scores of user u to these items in order to predict $f(u, i)$. In practice, item-based CF methods work better because similarity in items tends to exist more naturally than between users [14]. That is, items tend to cluster better under categories of similar items, whereas users are likely to have more unique tastes. A detailed presentation of traditional item-based CF algorithms can be found in [24].

Notice that in both approaches we can only use *known* interactions, so if the utility matrix is very sparse we may not be able to make reliable predictions or even make a prediction at all (e.g. when none of the neighbors have rated the item i for which we are predicting $f(u, i)$). Another related problem of this method is the infamous *cold-start problem*, where we cannot make a good (or any) prediction for new items (nor for new users) that have not yet been rated by enough users.

It's worth noting that this method can actually be seen as a generalization of a k-nearest-neighbor classifier [1], which is an instance-based classifier that delays training until inference (i.e. *lazy learner*).

Model-based methods

Model-based methods employ machine learning models (typically *eager learners*) in order to solve the prediction problem. This problem can be formulated as a supervised learning problem where we are trying to learn a function $f(x) = y$ from labeled data instances in the form (x, y), where y (i.e. the label) represents the utility matrix's cell values and x represents a user-item pair. In other words, the data that we have available to train the model are the *known* cells in the utility matrix (i.e. each cell is a single data instance), whilst the *unknown* cells are the data for which we want to infer their values.

Depending on the utility matrix's cell value types, this supervised problem can be that of binary classification for unary/binary y values or regression for arbitrary y values, such as ratings.

However, it is not straightforward how to use the utility matrix in order to represent a user-item pair x in a suitable way for an arbitrary machine learning model to train on (x, y) data instances. Is representing users as the rows and items as the columns in the utility matrix still the best choice? How do we represent a user-item *pair* and input it into a machine learning model? Would concatenating the two vectors into one suffice, or can the model take in pairs of separate vectors? Are missing values in such vectors a concern? All these are valid questions that a model-based method should answer. Of course, to a certain extent, the answer also depends on the type of machine learning model used.

Some methods avoid dealing with pairs altogether. For example, in [1] it is mentioned that one could *fix* one of the two - e.g. the items - as a dependent variable and train a *separate* machine learning model (e.g. decision trees) for each item using all the *known* cells from other columns. In this case, the other items (i.e. the other columns) are the features and the rows of the utility matrix are the data instances. The model of choice here should be able to handle missing values, as the utility matrix is naturally sparse.

Other methods, such as Latent Factor models [3], learn fully-specified *latent vector representations*, aka *embeddings*, for both users and items, thereby solving the missing values problem, and employ models that take in pairs of separate latent vectors, thereby bypassing the issue of representing a user-item pair as one vector. These latent vectors (aka *factors*) could be learned separately using dimensionality reduction methods on the utility matrix or directly whilst learning to complete the matrix, like in Matrix Factorization methods. By using latent representations, we do lose some explainability. However, these methods have been shown to perform significantly better [1]. Matrix factorization will be presented in more detail in Subsection 2.3.1.1.

2.2.2 Content-based methods

The term "content-based" can be used to characterize recommender systems that make use of auxiliary information for items (or even users as well). This information is usually task-specific features we have access to.

Content-based recommender systems try to match users to items that are similar to items that they have liked in the past. Unlike collaborative filtering methods, however, their approach is not based on interaction correlations across multiple users, but solely based on the attributes of items and, more specifically, those rated by the same user in the past [1, 17]. In other words, the interaction scores of *other* users play no role whatsoever and, thus, they are not needed to make recommendations to a certain user.

As a result, these methods can usually offer better personalization, e.g. for users with unique tastes, as well as avoid the cold-start problem for new items with few interactions. At the same time, however, it is required that we have access to relevant item features representing the content of the items as well as some recorded past interactions for the user at hand so that we can estimate his preferences in items (that is, unless we already have access to other user features for that purpose).

In a typical content-based recommendation system, with access to item features, that solves the ranking version of the recommendation problem we operate as follows:

1. First, we use auxiliary information for items to create an item profile for each item, i.e. a feature vector for it. These features are item-specific and they represent the "content" of an item. They usually need to be numeric for common similarity metrics to work, so categorical features should probably be one-hot or multi-hot encoded. They could even be representative dense vectors, i.e. *embeddings*, learned via representation learning.

2. After creating all the item profiles, we use the known interaction values (i.e. utility matrix cells) for user-item pairs in order to build a user profile for a user by *appropriately* aggregating the item profiles of the items that *he* has interacted with. That is, *we build user profiles from the item profiles*. These user profiles represent the preferences of users in the attributes of items.

   There is no free lunch concerning the choice of aggregation, but a common choice is using the average of item profiles when we have binary interactions and a weighted average when we are dealing with, for example, ratings [14].

3. Having constructed a user profile for a certain user u, we can rank all the items by descending similarity (e.g. cosine similarity) between said user profile and each item profile. We can then simply recommend the k most similar items for top-k recommendation.

   The similarity metric used between item and user profiles chosen should be something suitable for the features we used and the way we constructed the user profiles.

2.2.3 Hybrid methods

Hybrid methods attempt to inherit the best of both worlds by combining both sources of information. That is, they seek to exploit both the correlations between users and/or between items in the utility matrix as well as the user's preferences in item attributes.

According to [1, 5], we can build a hybrid recommendation system in a variety of ways including taking a weighted combination of an ensemble of different methods (e.g. like the Netflix prize winners did albeit this was a purely CF method), cascading the recommenders in a way where each refines the recommendations given by another [7], training a recommendation system on both sources of information directly [18, 25], etc.

2.3 Deep learning approaches

An insightful survey of deep learning methods applied to the recommendation task can found in [33]. In this paper, the authors go through many proposed deep learning architectures and briefly describe some of them. From these methods, Neural Collaborative Filtering [11] stands out to be a simple and flexible framework, as it can be used as both a pure collaborative filtering method and a hybrid method (by adding content-based profiles to it) and it can admit a variety of possible losses depending on our goal.

2.3.1 Neural Collaborative Filtering

Neural Collaborative Filtering is a method that extends Matrix Factorization (MF). We therefore present the basics of Matrix Factorization methods first.

2.3.1.1 Matrix Factorization

The basic idea of Matrix Factorization (MF) methods is to decompose the $m \times n$ utility matrix R into two matrices: an $m \times k$ matrix U and an $n \times k$ matrix V, such that:

$$R \approx UV^T \tag{2.1}$$

That is usually achieved by finding matrices U and V (e.g. via gradient descent) that minimize the approximation error as measured by the quantity [1]:

$$\|R - UV^T\|^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} |r_{ij} - \vec{u}_i \cdot \vec{v}_j|^2 \tag{2.2}$$

Assuming that there exist enough correlations in the utility matrix R, it is possible to use a small enough $k \ll \min(m, n)$ to achieve a small approximation error, even though R has missing values. To deal with missing values we can just exclude those terms from the sum in 2.2.

The k-dimensional row vectors of matrix U represent latent factors (i.e. *embeddings*) for *users*, while the k-dimensional row vectors of matrix V represent latent factors for *items*. These vectors represent the *affinity* of a user or an item towards the k latent *concepts* that are extracted from the utility matrix.

These *concepts* are usually arbitrary and do not have an interpretable semantic meaning (unless more restrictions are placed upon U and V e.g. non-negative matrix factorization). However, a useful example to understand the intuition behind these vectors is presented in Figure 2.3, where the concepts are genres of movies, a user factor is a vector representing if a user likes / is neutral towards / dislikes a genre and an item factor represents if a movie belongs or not to one or more genres.

Having access to these latent factors for users and items the prediction $\widehat{r_{ij}}$ of the model for the i-th user and the j-th item is defined as the dot product of the two latent factors:

$$\widehat{r_{ij}} = \vec{u}_i \cdot \vec{v}_j = \sum_{s=1}^{k} u_{is} v_{js} \tag{2.3}$$

Therefore, by minimizing 2.2 we minimize the margin of error (i.e. the mean squared error) in our predictions for the *known* cells of the utility matrix.

An interesting part about this process is that after finding U and V we can reconstruct the whole utility matrix in one shot.

Of course, more advanced variations have been proposed and used in practice [13]. These may involve adding regularization (e.g. Funk MF [8]), adding bias terms for users and items (e.g. biased SVD), etc. but the basic idea is the one presented above.

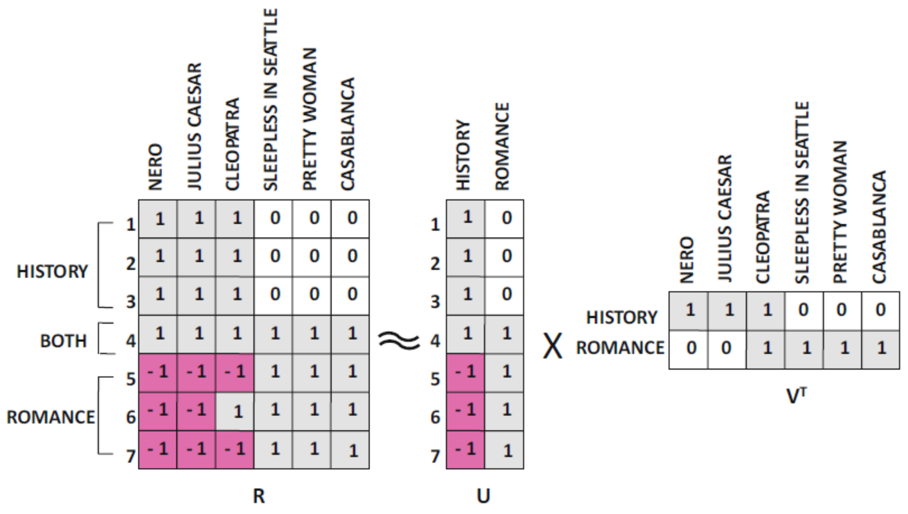


Figure 2.3. *A manufactured example of a matrix factorization for movies taken from [1].*

2.3.1.2 Neural Collaborative Filtering

The Neural Collaborative Filtering (NCF) framework is presented in [11] and is a generalization of Matrix Factorization methods. In this paper, the authors note that MF methods linearly combine the latent concepts in 2.3 and, as such, can be deemed as linear models of latent factors. They then describe how this, i.e. using the dot product of latent vectors for items and users as the model's prediction, can be limiting.

Instead of increasing the latent concepts k, which can increase the model's expressiveness but adversely hurt its generalization (e.g. by overfitting), the authors suggest the use of deep neural networks in order to model the interaction between user and item latent vectors rather than using a simple dot product. They argue that this can potentially make the model more expressive by being able to combine latent concept with different weights and by being able to capture non-linear user-item interactions.

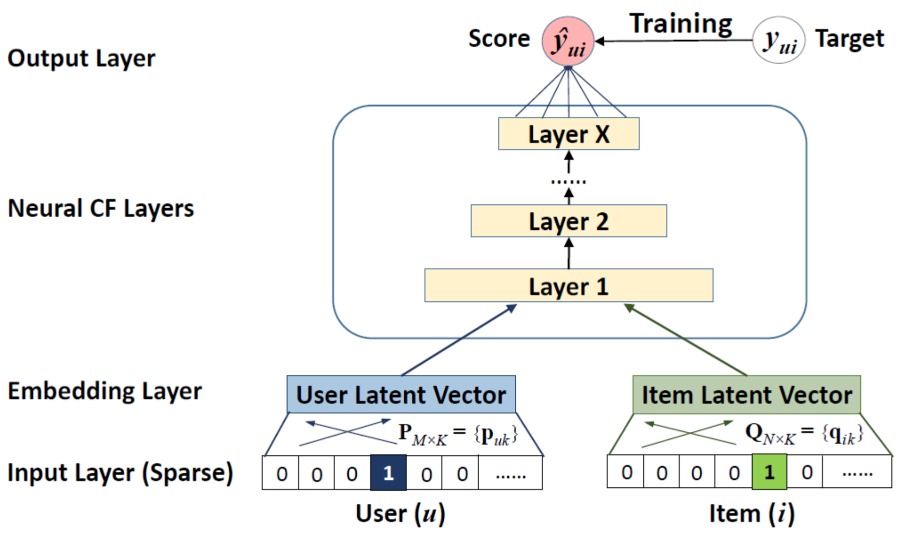


Figure 2.4. *The Neural Collaborative Filtering (NCF) framework.*

The basic network structure for this framework is summarized in Figure 2.4 and is made up of the following parts:

Input layer

The network has two separate vector inputs: one for a user u and one for an item i. This allows our network to directly model a preference score $f(u, i)$ between the user u and the item i by working on a user-item pair at a time.

The type of input vectors, $\vec{v_u}$ and $\vec{v_i}$, we use to represent users and items under this framework is extremely flexible. For example:

1. We can use one-hot vectors, if we don't have access to content (i.e. features) for items and/or users and we do not care about being able to generalize to new items and/or new users (we still want to generalize to unknown interactions i.e. user-item pairs). By doing this, we effectively memorize embeddings for the users and items we have access to.

2. We can still represent users as their rows and items as their columns in the utility matrix as in neighborhood-based methods [31]. This should allow our model to generalize to new users and new items, without having to retrain it, as we can represent new users by their interactions with *known* (only) items and new items by their interactions with *known* (only) users, assuming that there are any, of course. There may very well not be due to the cold-start problem.

3. If we do have access to say auxiliary item features, then we can use content-based methods in order to represent items with item profiles and users with user profiles. This would allow our network to generalize to new items and to new users, without the cold-start problem for items.

Embedding layer

For each input vector we use a separate embedding layer, i.e. a simple feed-forward layer, in order to learn to project items and user input vectors to a latent vector space of their own. So by using a $m \times k$ weight matrix P for users and an $n \times k$ matrix Q for items, we can get their embeddings by simply forwarding their input vectors as:

$$\vec{e_u} = P^T \vec{v_u} \quad \text{and} \quad \vec{e_i} = Q^T \vec{v_i} \tag{2.4}$$

Note that if we already have access to user or item embeddings as input vectors, we could theoretically skip this extra layer or keep it to tune them further. We could even add a non-linear activation like ReLU to force the embeddings to be non-negative (like in non-negative matrix factorization).

Neural CF layers

After acquiring a user and an item embedding, instead of simply taking their dot-product as in Matrix Factorization methods, we concatenate the two latent vectors and pass the resulting vector through a deep neural network, let's call it *MLP*, that the authors dubbed as neural CF layers. The role of these layers are to capture the potentially complex non-linear interactions between users and items.

Output layer

After passing through the neural CF layers we reach the output layer which has only one neuron outputting the prediction $\widehat{f}(u, i)$ or $\widehat{y_{ui}}$ of our network for the input user-item pair. If we are outputting probabilities, then this layer should have a sigmoid activation function, otherwise (e.g. for regression) no activation function is needed.

Put together, the output of the network (without the sigmoid) should be:

$$\widehat{y_{ui}} = \widehat{f}(u, i) = MLP(concat(\vec{e_u}, \vec{e_i})) \tag{2.5}$$

Loss function

This framework can be used to solve both the prediction and the ranking versions of the recommendation problem. To solve the prediction problem we would use a *pointwise* loss on (user, item, label) samples (u, i, y_{ui}) such as:

1. The Mean Squared Error (MSE) loss, if we have access to arbitrary interactions and we are performing regression. In this case:

$$L(u, i) = (y_{ui} - \widehat{y_{ui}})^2 \tag{2.6}$$

2. The Binary Cross-Entropy (BCE) loss, if we have access to unary or binary interactions and we are performing binary classification. In this case:

$$L(u, i) = -y_{ui} \log \widehat{y_{ui}} - (1 - y_{ui}) \log (1 - \widehat{y_{ui}}) \tag{2.7}$$

   Note that performing binary classification for binary targets is preferable because the loss ranges from 0 to ∞ instead of from 0 to 1. In this scenario, we should add a sigmoid activation function in our output layer in order for it to output probabilities.

To solve the ranking problem we would typically use a *pairwise* ranking loss, such as the Bayesian Personalized Ranking (BPR) loss, on (user, item, item) triplets (u, i, j), where we want item i to be ranked higher than item j for the user u. In this setting, which is also discussed in [26], we would need to forward our model twice for each triplet: once for (u, i) and once for (u, j). After doing so, we can define the BPR loss as:

$$L(u, i, j) = -\log \sigma(\widehat{y_{ui}} - \widehat{y_{uj}}) \tag{2.8}$$

where σ is the sigmoid function. This loss function is minimized when $\widehat{y_{ui}}$ is larger than $\widehat{y_{uj}}$ i.e. when we are indeed ranking item i higher than item j.

It is worth noting that the triplets used in pairwise learning are typically going to be a lot more than the user-item pairs in pointwise learning. If a user u has interacted with say n items, then there are $n(n-1)/2$ possible (i, j) pairs of items to form an equal amount of (u, i, j) triplets[1], while there are only n possible user-item pairs. This means that one may have to rely on some kind of sampling scheme and not examine all of them in each epoch. An example of such uniform sampling for binary interactions from implicit data is presented in [23].

It is also worth mentioning that the BPR loss has been criticized for suffering from vanishing gradients [22], where for correctly ranked item pairs the gradients can become very close to zero. This essentially means that we learn next to nothing from these updates. When employed with uniform sampling, this phenomenon slows down convergence as this kind of pairs waste calculations and it may take a while to draw useful ones. According to the authors of [22], this is especially true when the item popularity is tailed, e.g. it follows a Power-Law distribution[2]. To combat this, they suggest a context-dependent sampling strategy which oversamples the most informative pairs. That being said, simple random sampling approaches can still be considered viable, as shown in [29].

2.3.1.3 Neural Collaborative Ranking

The authors in [26] prefer a similar but more specialized (in terms of ranking) architecture, which they call Neural Collaborative Ranking (NCR). In this architecture, which is presented in Figure 2.5, they forward a similar model to concatenated embeddings for u, i and j and predict a preference score y_{uij} for the whole triplet directly, which expresses the user's preference of item i over item j. This score is then used directly inside the sigmoid of Equation 2.8. They claim that the intuitive advantage of this architecture over the previous one (i.e. NCF with BPR loss) is that it can also model the non-linear interactions between the two items.

However, predicting a score for the triplets instead of user-item pairs means that we cannot just rank the items' scores for top-K recommendations anymore. To solve this situation, the authors suggest a heuristic algorithm, based on the transitive property that the predicted scores of ranked items should ideally have.

2.3.1.4 Neural Matrix Factorization

The authors in [11] go one step further to suggest an architecture that combines both Matrix Factorization, or a more generalized version of it they refer to as Generalized Matrix Factorization (GMF), and Neural Collaborative Filtering (NCF). They call this architecture Neural Matrix Factorization (NeuMF).

[1]This estimate would assume that we also pair together items of the same ranking which is something we do not want to do e.g. when we have binary interactions we would only pair a positive with a negative item. So the actual number is less than this estimate e.g. $n_{positive} \times n_{negative}$ but still considerably larger than n.

[2]https://en.wikipedia.org/wiki/Power_law

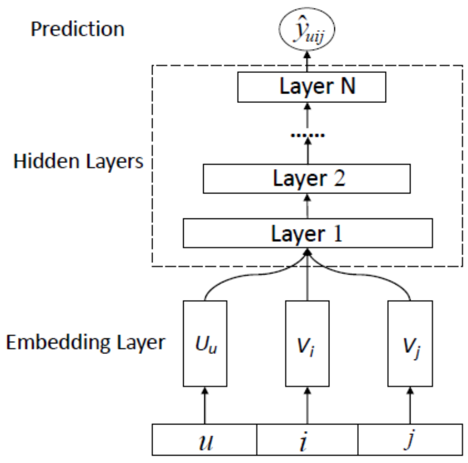


Figure 2.5. *The Neural Collaborative Ranking (NCR) framework.*

NeuMF is presented in Figure 2.6. It combines NCF and GMF by concatenating the result of their last hidden layer and adding a final output neuron after that. The idea is that, with such a model, we can use both a shallow and a deep representation of the user-item interactions to make our final prediction. In the authors' experiments, NeuMF is slightly better than NCF, which in turn is better than GMF.

Some things that the authors note are that:

1. It is better to use separate user and item embedding layers for the two models and not restrict them to share the same ones (e.g. GMF may need larger embeddings).
2. We can first train each subnetwork separately and then combine them by initializing the respective weights of NeuMF with their values in the pre-trained GMF and NCF networks.

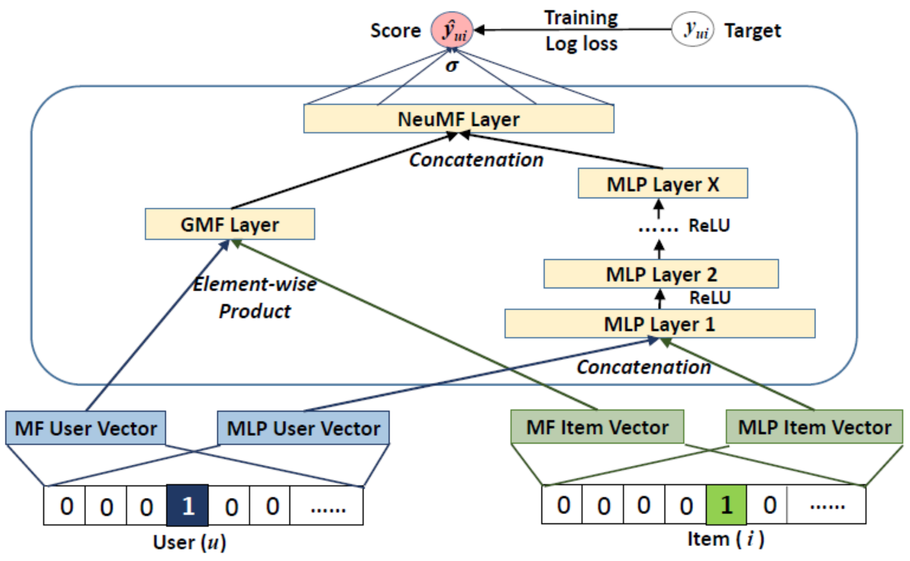


Figure 2.6. *The Neural Matrix Factorization (NeuMF) framework.*

2.3.2 Neural Graph Collaborative Filtering

A survey of Graph Neural Network (GNN) approaches to the recommendation setting can be found in [30]. The motivation behind these approaches is to learn better user and item embeddings by leveraging the structure of the bipartite user-item interaction graph and the relationships of multi-hop neighbors in it. From the many possible approaches here, Neural Graph Collaborative Filtering [29] and Light Graph Convolution Networks [10] appear to be a natural extension to the Neural Collaborative Filtering framework using GNNs. In order to present these methods, we first dive into the basics of Graph Neural Networks in general.

2.3.2.1 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural networks that work on graph data. Given a graph G, with an initial set of nodes V, each of which has an initial vector representation, GNNs *learn* how to use the structure of the graph (i.e. its edges) in order to acquire new latent vector representations or embeddings, aka *hidden states*, for each node in the same or a similar graph. These new representations depend on said structure as well as the initial representations. In other words, GNNs are just encoders for nodes on graphs that leverage the graph's structure[3], as shown in Figure 2.7.

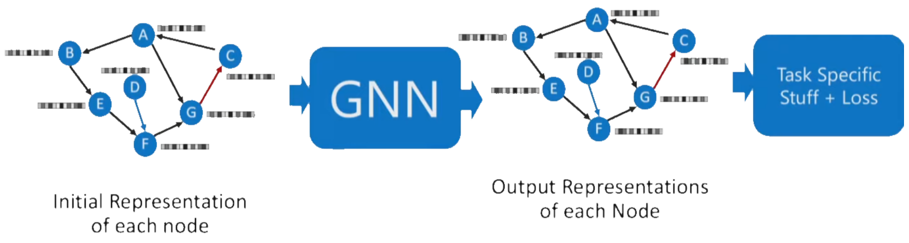

A
B
C
D
E
F
G
Initial Representation of each node
GNN
Output Representations of each Node
Task Specific Stuff + Loss

Figure 2.7. *Graph Neural Networks (GNNs) as encoders.*

In order to do this, they implement what is known as the *Message Passing Framework* [9]. In this framework, for each directed edge from node v to node u in the graph G we determine that a message $m_{v \to u}$, which is just a vector itself [4], will be passed from node v to node u. Via these messages, information is propagated through the structure of the graph. Each node sends messages to all his outgoing neighbors and receives messages from all his incoming neighbors in a series of time steps $t = 1, 2, ..., T$, where T is a hyperparameter of the process.

After the first iteration ($t = 1$) every node embedding contains information from its 1-hop neighborhood, after the second iteration ($t = 2$) every node embedding contains information from its 2-hop neighborhood, etc. In general, after the k-th iteration, every node embedding contains information from its k-hop neighborhood, as shown in Figure 2.8 for $k = 3$. Therefore, by using $T > 1$, we can model high-order connectivity in the

[3] https://www.microsoft.com/en-us/research/video/msr-cambridge-lecture-series-an-introduction-to-graph-neural-networks-models-and-applications/

[4] The message to be sent can be learned during training.

graph. At the same time, however, using too large a T can lead to a phenomenon known as *oversmoothing* [20, 6], where almost every node ends up passing information to every other node and, as a result, all the nodes end up with very similar embeddings. In any case, T should be tuned to the graph (e.g. performance on a validation set).

During each time step, the following operations take place:

1. First, each node v constructs and "sends" a message $m_{v \to u}$ to each of his outgoing neighbors $u \in N_v^-$.
2. Second, each node u "receives" and aggregates (e.g. average) its incoming messages $m_{v \to u}$ from each of his incoming neighbors $v \in N_u^+$.
3. Third, each node u updates its hidden state using his previous hidden state $h(u)$, or a self-message $m_{u \to u}$ created from it, and the aggregated incoming messages from his incoming neighbors.

Put together, as a general rule, for each time step $t = 1, 2, \ldots, T$ and for each node u we update its hidden state $h(u)$ as [9]:

$$h^{(t)}(u) = UPDATE\left(m_{u \to u}^{(t-1)},\ AGGREGATE\left(\{\, m_{v \to u}^{(t-1)} \mid v \in N_u^+ \,\}\right)\right) \tag{2.9}$$

where $h^{(0)}(u)$ is u's initial node representation, *UPDATE* and *AGGREGATE* are arbitrary differential functions and the messages $m_{v \to u}$ are usually created from the current hidden state of node v using a learnable feedforward layer, although more complex approaches can also be used (e.g. use u's hidden state as well, normalizations, etc.).

Based on the choice of *UPDATE*, *AGGREGATE*, the way we construct messages and the way we choose a final node representation (e.g. take the last hidden state, concatenate all T hidden states, aggregate them, use an RNN encoder like GRUs to learn what to keep from each hidden state, etc.), different GNN models have been proposed e.g. Graph Convolutional Networks (GCNs) [32], Gated Graph Neural Networks (GGNNs) [16], Graph Attention Networks (GATs) [27], etc.

2.3.2.2 Neural Graph Collaborative Filtering

The authors of [29] argue that methods such as Neural Collaborative Filtering, in which the collaborative signal is captured *implicitly* by using known user-item interactions in the objective function, are not sufficient to yield satisfactory embeddings for users and items. They claim that, because of this deficiency, these models have to rely on the interaction modeling, e.g. the neural CF layers of NCF, to make up for suboptimal user and item embeddings.

The authors propose that we ought to use the known user-item interactions in the embedding process for users and items as well, in order to *explicitly* capture the existing collaborative signals. To that end, they urge us to consider the bipartite graph of user-item interactions as shown in Figure 2.8. They suggest that, by using this graph as input to a GNN, we can acquire better user and item embeddings, which can then better contribute to the task of the overall network.

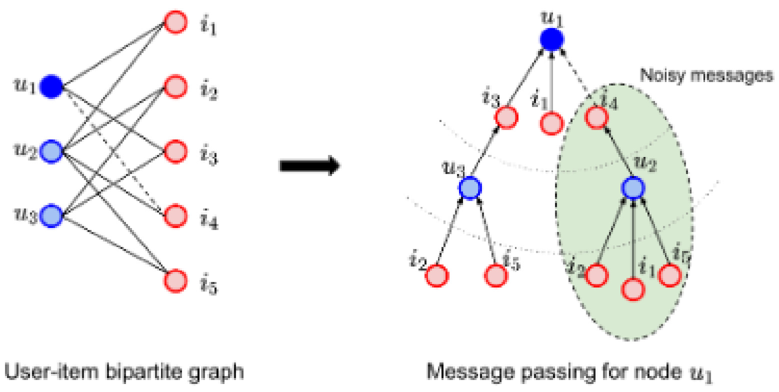


Figure 2.8. *An example of a user-item interaction bipartite graph and the message propagation on it for node u_1 for $T = 3$.*

The authors essentially suggest the same architecture as NCF, but with a GNN plugged in-between the embedding layers and the neural CF layers, that serves as an encoder which improves the user and item embeddings via message propagation on the user-item bipartite graph. In fact, they proceed to completely omit the neural CF layers and revert back to the simple dot product of MF methods for interaction modeling, as they argue that we can now learn good enough user and item embeddings to not need these extra layers. The architecture they used, which they call Neural Graph Collaborative Filtering (NGCF), is shown in Figure 2.9.

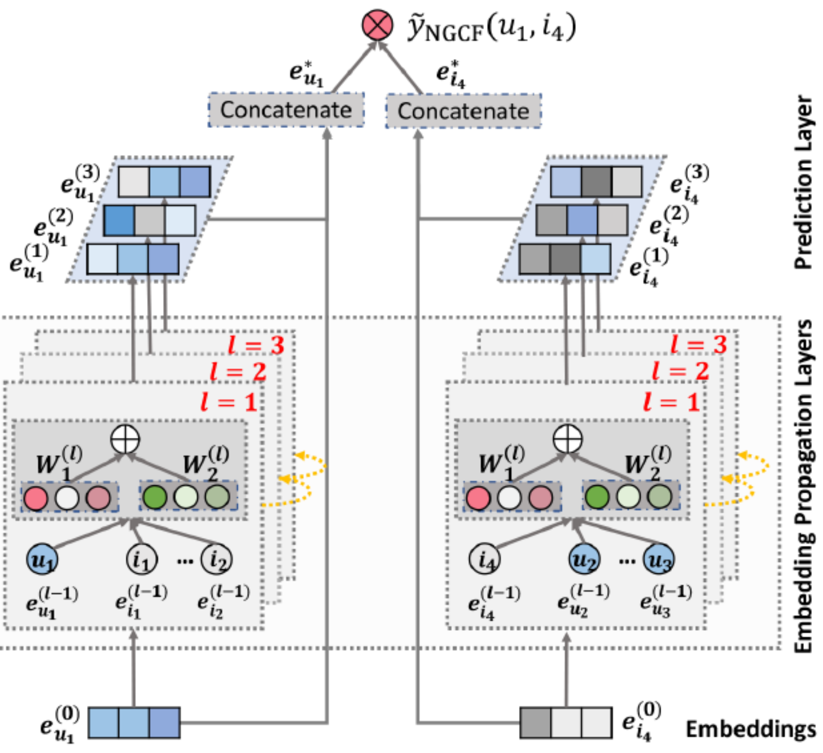


Figure 2.9. *The Neural Graph Collaborative Filtering framework (here $T = 3$).*

A caveat to keep in mind, which makes this method more costly to train than NCF, is that, in order to forward *one* user-item pair through the network, we have to run the GNN on the *entire* bipartite graph[5], since we need their neighbors and their neighbors' neighbors, etc. for message propagation. Consequently, if we are training the model with mini-batch gradient descent, we have to forward the entire graph into the GNN once per batch, unlike NCF where each batch's forward pass was independent of items and users outside of it.

We can summarize NGCF as follows.

Embedding layers

We first use the same embedding layers as in Equations 2.4 of NCF to acquire some initial embeddings for user and item input vectors. These layers can be trained end-to-end along with the rest of the network.

GNN layers

We then stack T (e.g. 2-5) GNN layers on top of these initial embeddings, that give us T new embeddings for both users and items by leveraging the structure of the user-item bipartite graph, as explained in Subsection 2.3.2.1. The authors made the following choices regarding the GNN implementation.

We construct a message $m_{u \to u}$ from node u to himself at time step t as:

$$m_{u \to u}^{(t)} = W_1^{(t)} \vec{e_u}^{(t-1)} \tag{2.10}$$

and we construct a message $m_{i \to u}$ from node i to a different node u at time step t as:

$$m_{i \to u}^{(t)} = \frac{1}{\sqrt{|N_i|\,|N_u|}} \left(W_1^{(t)} \vec{e_i}^{(t-1)} + W_2^{(t)} \left(\vec{e_i}^{(t-1)} \odot \vec{e_u}^{(t-1)} \right) \right) \tag{2.11}$$

where $\vec{e_i}^{(t)}, \vec{e_u}^{(t)}$ are the hidden states (i.e. embeddings) of nodes i and u respectively at time step t, N_i and N_u are their first-hop neighbors, $W_1^{(t)}, W_2^{(t)} \in \mathbb{R}^{d^{(t)} \times d^{(t-1)}}$ are learnable weight matrices for the t-th GNN layer, $d^{(t-1)}$ and $d^{(t)}$ are the embedding sizes associated with those layers and the $\odot$ operator signifies element-wise vector multiplication.

Notice that, unlike standard GCNs, the authors chose to also encode the user-item interaction between $\vec{e_i}$ and $\vec{e_u}$ into the messages being passed between user and item nodes via the term $\vec{e_i} \odot \vec{e_u}$. They claim that this has an important impact in the model's performance, as it makes the messages being passed depend on the affinity between the interacting user-item pairs.

To aggregate all the messages in a node u, including the one he sends to himself, we simply sum them (the messages are already normalized to account for node degrees) and, to get the new hidden state for u, we pass them through *LeakyReLU*:

$$\vec{e_u}^{(t)} = \mathit{LeakyReLU}\left(m_{u \to u}^{(t)} + \sum_{i \in N_u} m_{i \to u}^{(t)} \right) \tag{2.12}$$

[5]Or a subgraph containing all their T-hop neighbors

By using *LeakyReLU* as the activation function we allow messages to encode both positive and small negative signals whilst still introducing non-linearity, which is what makes deep learning powerful.

The authors also express the whole propagation rule in Matrix Form, so that we can forward the model efficiently, as:

$$E^{(t)} = LeakyReLU\Big((L+I)\,E^{(t-1)}\,W_1^{(t)} + L\,E^{(t-1)} \odot E^{(t-1)}\,W_2^{(t)}\Big) \tag{2.13}$$

where $E^{(t)} \in \mathbb{R}^{(N+M)\times d^{(t)}}$ are the embeddings for the N users and M items at time step t (with $E^{(0)}$ being their initial ones after the embedding feedforward layers), $W_1^{(t)}, W_2^{(t)} \in \mathbb{R}^{d^{(t-1)} \times d^{(t)}}$ are the transposed weight matrices of Equations 2.10 and 2.11, I is the $(N+M)\times(N+M)$ identity matrix and L is the Laplacian matrix of the same dimensions for the user-item graph and is constructed as:

$$L = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \quad \text{and} \quad A = \begin{bmatrix} 0 & U \\ U^T & 0 \end{bmatrix} \tag{2.14}$$

where A is the $(N+M)\times(N+M)$ adjacency matrix, $U \in \mathbb{R}^{N\times M}$ is the utility matrix, 0 signifies the all-zero matrix and D is the diagonal degree matrix (i.e. $D_{ii} = |N_i|$). The result is that L is a matrix with the same nonzero elements as the adjacency matrix A but with each nonzero (off-diagonal) entry normalized as $L_{ui} = 1/\sqrt{|N_u|\,|N_i|}$ as in Equation 2.11. This normalization "trick" was suggested by Kipf et al. [12].

Prediction layers

After applying the GNN propagation rules for T time steps, we acquire embeddings $E^{(t)}$ for $t = 1, 2, \dots, T$ for all users and items in the graph. From these embeddings, we would only keep the ones concerning user-item pairs we want to make a prediction for (e.g. the ones in the batch).

However, as briefly mentioned in Subsection 2.3.2.1, there are many possible choices on how to pick a final embedding for each node from the T embeddings we have acquired for them e.g. keep only the last hidden state, aggregate all T hidden states, use an LSTM on the sequence they create, etc. The authors opt for the simple approach of simply concatenating all the T node embeddings to form a final one:

$$\vec{e_u} = concat\Big([\,\vec{e_u}^{(t)} \mid t = 1, 2, \dots, T\,]\Big) \tag{2.15}$$

They argue that this is effective because it uses all the representations, which reflect different orders of connectivity in the graph, in a very simple way.

After having formed a final embedding for a user and an item, the authors make a prediction for the user-item pair using their dot product, as in traditional MF methods:

$$\widehat{y_{ui}} = \widehat{f}(u, i) = \vec{e_u} \cdot \vec{e_i} \tag{2.16}$$

Of course, we are free to add a sigmoid activation if, for example, we are training with a BCE pointwise loss for binary classification. We are also free to apply the same methods that NCF does for interaction modeling, i.e. concatenate the two embeddings and pass them through a series of neural CF layers before making a prediction as in Equation 2.5.

The same loss approaches as presented in Subsection 2.3.1.2 apply here as well, as this is essentially the same approach with NCF but with an extra GNN layer at the embedding process. The authors, for example, train their model using the BPR loss for ranking.

Message and node dropout

In order to make the model more robust against the existence or absence of single user-item interactions (i.e. edges) or certain nodes in the graph, the authors suggest the use of *message dropout* or *node dropout* respectively [4]. This is paramount if we want our model to generalize well to new graphs (or modified versions of the original one), as our model is very likely to overfit the input graph's structure without it.

Message dropout involves randomly blocking a percentage p (aka dropout rate) of edges in the graph so that messages are not passed through them during message propagation. To do this, we can randomly zero out p percent of the non-zero entries of the Laplacian matrix L. Node dropout consists of randomly blocking *all* messages of a percentage p of nodes during each propagation step. To do this, we can temporarily zero-out *all* the outgoing edges in the Laplacian matrix L for a random percentage p of nodes during each time step t. The authors of NGCF experimented with both methods and concluded that node dropout was consistently superior in their tests.

2.3.2.3 Light Graph Convolution Networks

Following up on the work of NGCF [29], the authors of [10] introduce a similar but much simpler GNN architecture that they name *LightGCN*. After performing an ablation study on NGCF's components, they empirically reached the conclusion that NGCF is too complex for the task of Collaborative Filtering[6], as many of its components (e.g. feature transformations, non-linear activation, etc.), that theoretically make it more expressive and have proven effective in other GNN applications, ended up not helping the model generalize better in their experiments. In fact, they found that removing them even improved the performance of the resulting model by as much as 16% on average under the same experimental setting [10].

In their new proposed model, LightGCN, they get rid of both learnable feature transformation layers W_1 and W_2 in Equations 2.10 and 2.11, completely remove self messages and instead define a message $m_{i \to u}$ from node i to a different node u at time step t to be just the normalized embedding of node i as:

$$m_{i \to u}^{(t)} = \frac{1}{\sqrt{|N_i|\,|N_u|}} \vec{e}_i^{\,(t-1)} \tag{2.17}$$

[6] They examined the task of pure Collaborative Filtering on implicit interactions.

They aggregate these messages in the receiver node u by simply summing them and they update the next hidden state of u with no self messages and without an activation function as:

$$\vec{e_u}^{(t)} = \sum_{i \in N_u} m_{i \to u}^{(t)} = \sum_{i \in N_u} \frac{1}{\sqrt{|N_i|\,|N_u|}} \vec{e_i}^{(t-1)} \tag{2.18}$$

This means that the only learnable parameters of the process are the initial embedding layers P and Q in Equations 2.4. Only they are responsible for learning suitable user and item embeddings that are then refined using the graph's structure.

In order to get a final embedding for node u after T GNN layers, instead of concatenating all the T hidden states as in NGCF, they compute a weighted average of each hidden state *and* the original embeddings as:

$$\vec{e_u} = \sum_{k=0}^{T} a_k \vec{e_u}^{(k)} \tag{2.19}$$

where $\vec{e_u}^{(0)}$ is the initial user or item embedding produced from the embedding layers in Equations 2.4 and the weights a_k sum to 1 and determine how much importance to give each GNN layer in the aggregation. It is because of the inclusion of the initial user and item embeddings $\vec{e_u}^{(0)}$ that the authors claim that self messages are not necessary in the updated hidden state. The weights a_k could be learned (e.g. via another differentiable component of the network), but the authors claim that setting each layer to be equally important, as in $a_k = \frac{1}{T+1}$ (i.e. using the standard arithmetic mean), generally works well and adds to the simplicity of their model.

After acquiring the final node embeddings, they also use the dot product to make a prediction for a user-item pair, as in Equation 2.16. As for the loss function, they also went for the BPR loss with uniform negative sampling, even though they point out that more advanced negative sampling methods exist that promise to speed up convergence. Last but not least, they mention that they did not use either message nor node dropout, as they deemed it unnecessary due to their model's simplicity compared to NGCF.

All in all, LightGCN is a much cheaper and lighter model compared to NGCF that, in these authors' experiments, even outperformed NGCF for pure Collaborative Filtering on implicit user-item interactions. The authors attribute its success to its simplicity. They argue that, when using one-hot vectors as input for items and users (as is the case with pure Collaborative Filtering tasks), learnable feature transformation layers are not as helpful as when they are used on rich node feature vectors (e.g. on other GNN applications). As a result, including such layers only hinders the learning process. Instead, they only keep the most essential graph operation, message aggregation, and they show that this is enough to achieve better performance, when we attempt to capture a – potentially limited – collaborative signal in the user-item bipartite graph.

Chapter 3

Methodology employed

In the context of this dissertation, we explore the application of the deep learning methods discussed in the previous chapter and consider possible extensions and modifications to them. One such extension we seek to employ is the use of content-based methods in order to create meaningful item and user profiles, instead of relying on plain one-hot vectors. This essentially converts these otherwise purely Collaborative Filtering methods to hybrid ones, which come with some of the added benefits that content-based methods are known for, such as avoiding the cold-start problem for new items and new users. We experiment with three major neural network architectures, all based on the original Neural Collaborative Filtering framework. The first one is the original NCF, the second adds an item-item attention mechanism to the construction of user profiles, while the third introduces Graph Neural Networks in the user and item embedding process. Last but not least, we investigate two of the most popular training formulations for these kinds of models: training them for the prediction problem using a pointwise loss versus training for the ranking problem directly using a pairwise loss.

3.1 Content-based profiles

3.1.1 Item Profiles

There is a plethora of ways to create item profiles depending on the type of features we have access to. Perhaps the most common features we could have access to are simple numerical or categorical features. As mentioned in [14], there is a straightforward way to covert and aggregate these kinds of features into a purely numerical feature vector that is suitable for input to ML models like neural networks:

1. For numerical features (e.g. the duration of a movie), we can just take their value and place it in one dimension of the item profile vector. It is also very common to normalize all the numerical features (e.g. to $[0, 1]$ or $[-1, 1]$) so that they are in the same scale.

2. For categorical features (e.g. the genre(s) of a movie), we one-hot (if the values are mutually exclusive) or multi-hot (if they are not mutually exclusive) encode them. That is, we convert them to a zero vector, where only the values that exist have a value of one. Thus, just one categorical feature will end up taking up multiple

dimensions (as many as all its possible values) in the item profile vector, many of which might be zero, thereby making the vector sparse.

A useful property of such built item profiles is that cosine similarity can be used as a meaningful measure of similarity between items. In fact, this is what content-based methods as described in Subsection 2.2.2 rely on to make top-k recommendations.

If we were to apply a traditional content-based method, we might also want to multiply all entries associated with a feature by a weight (as a hyperparameter)[14], with which we could tune how much importance to assign to each feature. However, as we intend to use these profiles as input to a neural network, we can just let the neural network decide itself which feature to give more importance to, by ending up with larger weights for it.

It is worth mentioning, that we could also have access to content in the form of text or image. For text content, we could choose between sparse bag-of-word representations like Tf-Idf[1] and (pre-trained) NLP models (like BERT[2]) to extract meaningful embeddings from it. For image content, we could use (pre-trained) convolutional neural networks to extract embeddings from them. We could then concatenate all these embeddings to a single vector as the item profile, or have several item profiles for each item.

3.1.2 User Profiles

As mentioned in Subsection 2.2.2, in order to create a user profile for a user u that has already interacted with certain items $i_1, i_2, \dots, i_{|N(u)|} \in N(u)$, we typically aggregate the item profiles of those items in a suitable way so that they reflect the user's preferences in item attributes. The idea is to learn to make similar recommendations to users with similar preferences in item attributes by making those users have similar user profiles.

Once again, it should be noted that there is no one-beats-all approach to this and one could devise multiple approaches in an attempt to better capture user preferences, depending on the interaction types that are available and/or any auxiliary information about the user.

A distinction can be made between using fixed versus dynamic user profiles. In the first case, fixed user profiles are built from fixed item profiles, based on the user's known interactions. Note that this needs only happen once, before any training takes place. In the second case, we incorporate user profile construction into the forward pass of the network. While this is more expensive, there are certain advantages to it, as we shall see.

Fixed user profile construction

If we are dealing with unary interactions, then perhaps the most natural choice for building a user profile would be to simply average all the interacted item profiles (i.e. those with a utility matrix value of 1) as:

[1] https://en.wikipedia.org/wiki/Tf-idf

[2] https://en.wikipedia.org/wiki/BERT_(language_model)

$$\vec{v_u} = \frac{1}{|N(u)|} \sum_{i \in N(u)} \vec{v_i} \tag{3.1}$$

where $\vec{v_u}$ is the user profile and $\vec{v_i}$ are the item profiles. In regard to this aggregation, we note that:

1. Averaging item features like this can work well for the numerical features of said items if they follow a suitable (in that the mean of the distribution can be estimated by taking the arithmetic mean) type of distribution, e.g. a Gaussian one. But it could also not work well if say a user is split between two extremes e.g. liking only short movies or extremely long ones, in which case, by using the average movie length in the user profile, we would falsely infer that he likes medium-length ones. Our hope using this method is that the former is more common than the latter.

2. As far as categorical features go, taking an average simply gives us the percentage of interacted items with a certain categorical value. For instance, if 50% of the movies a user has watched were comedies whereas only 10% were dramas, this would mean that the user, on average, prefers comedies and this will be reflected in his user profile.

If we are dealing with binary or (mostly) explicit interactions such as arbitrary ratings (e.g. 1-5) then, while we could reduce this to the unary case[3], we can do something better. We can leverage the fact that we now have a positive and a negative preference score by taking a weighted average of the item profiles, where highly rated item profiles get a positive weight and lowly rated item profiles get a negative weight. We can do this via the following weighted aggregation:

$$\vec{v_u} = \frac{1}{|N(u)|} \sum_{i \in N(u)} (r_{ui} - \overline{r_u})\, \vec{v_i} \tag{3.2}$$

where r_{ui} is the rating of user u to item i and $\overline{r_u}$ is the user's *neutral* rating to items. For this neutral rating $\overline{r_u}$ we could use:

1. The arithmetic mean of all the possible ratings, e.g. if we have ratings from 0 to 5 we can use $\overline{r_u} = 2.5$. This reasonably assumes that a user likes anything rated above this number and dislikes everything below it.

2. The mean rating that the user rates items with as in $\overline{r_u} = \frac{1}{|N(u)|} \sum_{i \in N(u)} r_{ui}$. By using this, we account for different users being more or less strict with their ratings, but we assume that a user has rated both items that he likes and dislikes about equally, which may not always be true.

3. We can combine both previous options by taking their mean in order to be somewhere in the middle. This is what we found to work the best empirically.

[3]By applying a binary threshold and looking only at positive interactions.

The idea is that now we can have similar user profiles for users that, not only like the same item attributes, but that also dislike the same item attributes. The latter attributes will usually take negative values in the user profiles. For example, if a user dislikes thrillers, then we expect him to have rated them less than his neutral rating, resulting in a negative thriller value in his user profile. On the other hand, if he loves dramas, then we similarly expect a positive drama value. Furthermore, we can have more accurate user profiles by accounting for the degree of liking and disliking an item, which is available to us from the known explicit interactions.

While this aggregation makes sense for categorical features that are one-hot or multi-hot encoded as well as for dense embeddings, it is perhaps not as sensible when we blindly apply it to numerical features such as the duration of a movie. To see that, imagine that user u_1 has highly rated long movies, while user u_2 has both highly rated long movies and lowly rated short movies. We would expect user u_2 to have a higher user profile value for the movie duration than user u_1 (because of the opposite nature of short vs long movies), however the opposite would happen if we were to use Equation 3.2, because of the summation of extra negative terms. To capture this opposite nature, we should normalize positive numerical features like movie duration to $[-1, 1]$ instead of $[0, 1]$, as this fixes this issue.

Dynamic user profile construction

Having a fixed user profile as explained above for a user with certain item interactions has its advantages, mainly the fact that it's an efficient calculation that can be done independently and prior to training our neural network. In that calculation, however, every interacted item is given the same weight of $1/|N(u)|$ without taking into account the candidate item for which we are trying to make a prediction (recall that all our networks make predictions for user-item pairs).

The authors of [28], who use a similar architecture to that of NCF, make use of an item-item attention mechanism in which, instead of assigning the fixed weight of $1 / |N(u)|$ in each interacted item, we *learn*[4] the weight to use via a secondary neural network which they dub *AttentionNet*. This network takes as input the concatenated item profiles or the item embeddings of the candidate item and each interacted item (one-by-one) and outputs an *attention* score. After outputting all the attention scores, we normalize them by passing them through a softmax activation layer so that they sum to 1. Now we are able to use these weights instead of the fixed $1/|N(u)|$ weights in Equations 3.1 and 3.2 as:

$$\vec{v_u} = \sum_{i \in N(u)} a_{ci}\, \vec{v_i} \tag{3.3}$$

$$\vec{v_u} = \sum_{i \in N(u)} a_{ci}\, (r_{ui} - \overline{r_u})\, \vec{v_i} \tag{3.4}$$

where c is the candidate item, a_{ci} is the normalized attention weight in $[0, 1]$ for the interacted item i based on the candidate item c and $\sum_{i \in N(u)} a_{ci} = 1$.

[4]In an end-to-end fashion, during the training of the whole network.

It is worth noting that the authors of [28] construct the user profile from item embeddings instead of item profiles, but they are forced to do so as they are dealing with text content. We are free to use the item profile vectors directly *or* the item embeddings from the item embedding layer of our neural network.

The idea behind this item-item attention mechanism is for our network to learn to which past interactions to pay more attention when making a prediction for a new interaction. It should potentially learn different attention weights for different candidate items and, as such, we would be using a dynamically different user profile for the same user depending on the candidate item for which we want to make a prediction.

This item-item attention mechanism should not only potentially make our model more expressive but also add some much desired explainability, where we can reason as to why certain items were recommended by visualizing the attention weights of previously interacted items. At the same time, however, dynamically aggregating item profiles during the forward pass is significantly more computationally expensive than using a precalculated user profile.

3.2 Examined models

3.2.1 Basic NCF

The first and most basic category of models we sought to try, which we shall hereby refer to as Basic NCF, was that of the simple Neural Collaborative Filtering, presented in Subsection 2.3.1.2, using fixed input vectors for users and items, as shown in Figure 3.1. In Basic NCF, we simply forward the user and item vector inputs into their respective embedding layers (to which we have added bias terms but no activation function[5]) to get the embeddings that will be forwarded to the MLP network, which ultimately makes a prediction.

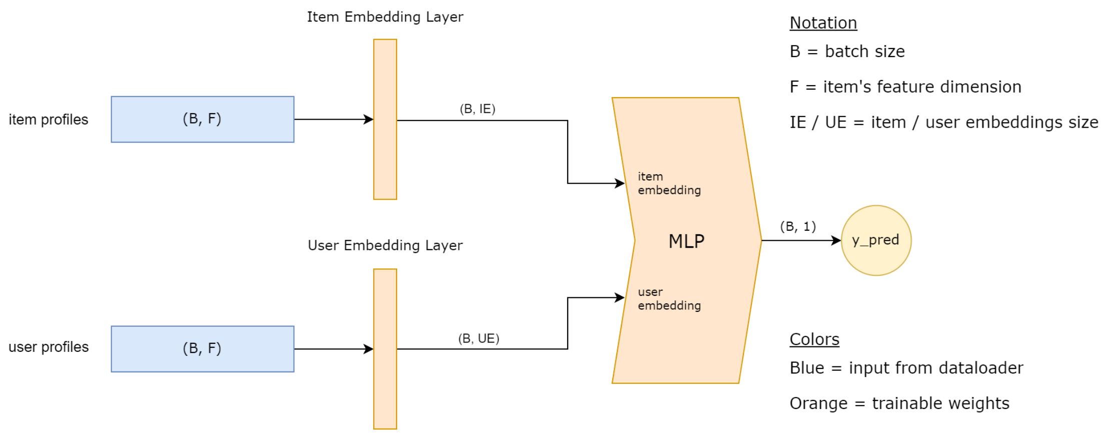


Figure 3.1. *The Basic NCF architecture.*

[5]Adding an activation function like ReLU hurt the model's performance in our experiments.

There are three kinds of inputs we experimented with:

1. Using one-hot vectors for items and users.

2. Using fixed content-based item profiles for items, but one-hot vectors for users.

3. Using fixed content-based item and user profiles.

The first amounts to a pure Collaborative Filtering approach, the second tests to see if item profiles alone help our task and the third tests if the constructed user profiles along with the item profiles help our task.

In this case, since the user profiles are fixed, they are constructed once, in the beginning. This makes this model faster to train, since we do not need to construct any user profiles during training. However, we would still need to construct the user profile for a new user during inference.

3.2.2 Attention NCF

The second category of models we examined, which we shall refer to as Attention NCF, is an extension of the Basic NCF architecture that constructs user profiles dynamically during the forward pass with an item-item attention mechanism as in Equation 3.4. The idea of Attention NCF is to learn to which rated items to give more attention, depending on the candidate item we are trying to predict the preference score of.

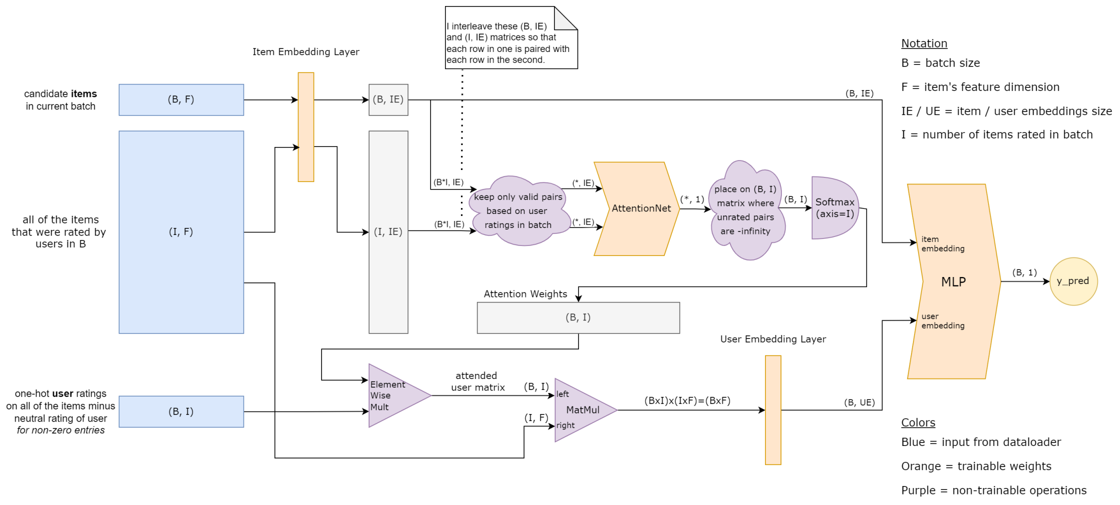

I interleave these (B, IE) and (I, IE) matrices so that each row in one is paired with each row in the second.
Item Embedding Layer
candidate items in current batch
(B, F)
(B, IE)
(B, IE)
all of the items that were rated by users in B
(I, F)
(I, IE)
(B*I, IE)
(B*I, IE)
keep only valid pairs based on user ratings in batch
(*, IE)
(*, IE)
AttentionNet
(*, 1)
place on (B, I) matrix where unrated pairs are -infinity
(B, I)
Softmax (axis=I)
Attention Weights
(B, I)
User Embedding Layer
one-hot user ratings on all of the items minus neutral rating of user for non-zero entries
(B, I)
Element Wise Mult
attended user matrix
(B, I)
left
(I, F)
right
MatMul
(BxI)x(IxF)=(BxF)
(B, UE)
item embedding
user embedding
MLP
(B, 1)
y_pred
Notation
B = batch size
F = item's feature dimension
IE / UE = item / user embeddings size
I = number of items rated in batch
Colors
Blue = input from dataloader
Orange = trainable weights
Purple = non-trainable operations

Figure 3.2. *The Attention NCF architecture.*

Input

This network expects a different input from the data loader compared to the much simpler Basic NCF, as shown in Figure 3.2. Let B be the batch size, F the dimension of the item profiles, I the (unique) number of (sorted) rated items by users in the current batch and IE and UE the dimension of item and user embeddings. We expect:

1. A $B \times F$ matrix C with the item profiles of all candidate items (i.e. the items we are predicting a preference score for) in the batch.

2. An $I \times F$ matrix R with the item profiles of all rated items in the batch (i.e. items with known ratings from users in the batch).

3. A $B \times I$ matrix U that contains the terms $r_{ui} - \overline{r_u}$ of Equation 3.4 for each user u in the batch and for each rated item i (unrated items have a value of zero).

Calculating the item-item attention weights

There are a lot of ways we could calculate the attention weights for item pairs in the dynamic user profiles of Equation 3.4. As we already mentioned in Subsection 3.1.2, the authors of [28] do this based on item-item pairs via a secondary neural network that they call *AttentionNet*. The AttentionNet takes as input pairs of item embeddings and outputs a score. The bigger the score (compared to others), the bigger the attention weight will be (after the softmax normalization) for this specific pair of items.

We implemented this idea as depicted in Figure 3.2. First, we acquire the item embeddings for the item profiles in matrices C and R by passing them through the item embedding layer. Then, we interleave the resulting matrices' rows so that each row from the first is paired with each row from the second. One way for us to get an attention score for each item-item pair is to pass the two $(B * I) \times IE$ matrices through the AttentionNet and then reshape the resulting vector into a $B \times I$ matrix: B candidate items, I rated items. However, the user of each sample in the current batch will probably not have rated every item in R (different users have rated different items) and, therefore, this approach would unnecessarily calculate many attention scores for invalid item-item pairs, which should be masked out before applying softmax afterwards[6]. Instead, in Figure 3.2, we only calculate the attention scores for valid item-item pairs and then correctly place them in a $B \times I$ matrix, where every other cell is $-\infty$ so that their value after applying softmax will be zero. Finally, we apply softmax on the I axis of that matrix (i.e. the rows sum to 1) in order to acquire the final attention weights.

While this works reasonably well, there also exist other *differentiable* ways to learn the item-item attention weights of Equation 3.4. One other way is to use the cosine similarity of the item embeddings (or some other differentiable vector similarity measure e.g. dot product), instead of the output of a neural network like AttentionNet, as the attention score that we then normalize via softmax. By doing this, we essentially try to force the network to learn more meaningful item embeddings, whose similarity plays a direct role on how we dynamically create user profiles.

Last but not least, for both methods discussed, we could fall back on using the item profiles directly instead of item embeddings as inputs. For the first, this would allow AttentionNet to learn completely different things from the item profiles, but it would also make the whole operation much more expensive, since the item profiles will typically be much larger than their embeddings. For the second, it would mean that attention

[6]We had initially followed this approach and found it much slower than the latter.

can not be learned, but is instead fixed based on item profile similarity. While this still makes sense (because item profile similarity makes sense for meaningful item profiles), it is probably too restrictive and also much more expensive.

Constructing user profiles

Having calculated the attention weights in a $B \times I$ matrix A, we can implement Equation 3.4 with two steps. First, we perform an element-wise multiplication of matrices A and U in order to add the attention weights a_{ci} to the $r_{ui} - \overline{r_u}$ terms. Second, we do a matrix multiplication between that $B \times I$ result and the rated item profiles $I \times F$ matrix R. This results in a $B \times F$ matrix with the user profiles of the B users in the batch. After having dynamically constructed the user profiles for the users in the batch, we can proceed to pass them through the MLP network as in regular NCF.

Again, we could use the $I \times IE$ item embeddings instead of the item profiles in R and doing so gives very similar results, but we preferred using the fixed item profiles directly in this case, as it more closely resembles Basic NCF's user profiles.

Important caveat about the item profiles used in the user profiles during training

In order to maximize the training data (i.e. user-item interactions) we have available for training, we ended up using the same user ratings to items for both constructing the user profile and as training objectives. That has the unwanted side effect of using the very thing we are trying to predict as part of the input, even if it is somewhat encapsulated in the user profile.

When the weight of a rated item is fixed at $1/|N(u)|$ as in Equation 3.2, which is used for Basic NCF, this probably does not significantly hurt the model's generalization, as its role in the final user profile is limited (i.e. it is averaged out). If we want to use fixed user profiles, then it is also kind of unavoidable, unless we use completely different interactions for input and as objectives, which would adversely lead to fewer data for training.

When we attempt to learn the weight of rated items (as we do in Attention NCF), however, the fact that the candidate item always exists in the rated items as well can quickly lead to *overfitting*, as the network learns to always give all its attention to that item alone, in order to drastically minimize its training loss. Unfortunately, the network will never have access to the rating of a new item when it will be called to predict its preference score during inference, so this plan cannot generalize at all to unknown interactions, which is what we are ultimately interested in.

Therefore, to avoid this overfitting and *only during the training phase*, we explicitly set the attention weight of zero to item-item pairs that concern the same item[7] during the forward pass, thereby ignoring the rating to the candidate item. This solves the overfitting problem and ensures that we never use as input a user-item interaction that we are trying to predict during training.

[7]It is guaranteed that exactly one will exist per user in the batch.

Message dropout

We can actually take the previous idea one step further and randomly ignore $p\%$ (e.g. 10%, 20%, etc.) of the interactions of the users in each batch during the training phase, similarly to how message dropout works in GNNs. That is, during each forward pass of the training phase, we set an attention weight of zero to random rated items in matrix R, so that they are ignored during dynamic user profile construction (which very much resembles neighbor aggregation in GNNs). The idea is that, even if we were missing a small $p\%$ of the interactions of a user, the same recommendations should be made to him since he is still the same user. The noise introduced in this way should help regularize the model and make it more robust, especially if we want to use it on new users with unseen user profiles during inference.

3.2.3 Graph NCF

The third category of models that we examined was those based on Graph Neural Networks (GNNs). These models extend the architecture of Basic NCF by further tuning the initial user and item embeddings via a GNN as an encoder over the user-item bipartite graph. That is, they start by separately embedding the same fixed item and user profiles that Basic NCF does, then they enrich them by explicitly capturing the collaborative signal in the graph via multiple graph convolution layers, before finally inputting them into the MLP network to output the model's prediction. We also tried using a simple dot product instead of the MLP, as [29] and [10] do, but it led to worse generalization in our experiments. These operations are presented in Figure 3.3.

One caveat is that the graph convolutions have to be applied on the entire graph, so we need to first embed *all* user and items in the graph and then select only those in the current batch. Alternatively, if we use T graph convolution layers, then we theoretically need only keep the T-hop neighbors of the users and items in the current batch, since messages from more distance neighbors will never reach them. This may be something to consider if the graph is too large and if these T-hop neighbors consistently end up being significantly fewer than the full graph's nodes.

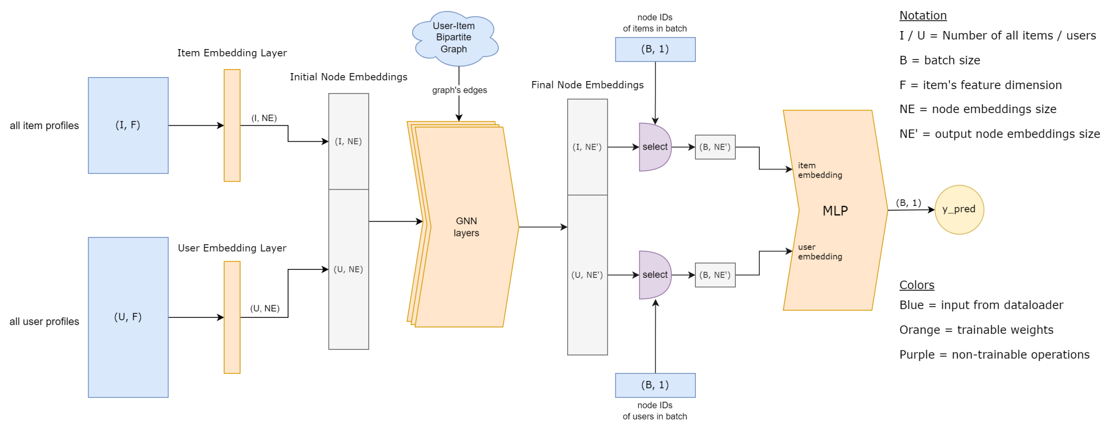


Figure 3.3. *The Graph NCF architecture.*

The user-item bipartite graph

The user-item bipartite graph is a graph consisting of user and item nodes where there exists a bidirectional edge (i.e. two directed edges) between a user node and an item node if and only if there is a known interaction between them. If we were dealing with unary or binary interactions, then this would suffice. However, assuming we have access to explicit user-item interactions such as ratings, we need to somehow take advantage of the rating of each interaction. The most straightforward way to do this is to assign each edge a weight based on its respective rating, and then appropriately leverage this weight when performing the graph convolutions.

Drawing inspiration from the way we aggregate item profiles in Equation 3.2, we thought it would be best to use the same formula for the edge weights: $r_{ui} - \overline{r_u}$ for user-to-item edges and $r_{ui} - \overline{r_i}$ for item-to-user edges. The idea is that edges will have a scaled positive or negative weight depending on if and how much their respective interaction is positive or negative.

An alternative choice would be to apply a binary threshold on each edge and only keep the edges for which $r_{ui} \geq \overline{r_u}$ and $r_{ui} \geq \overline{r_i}$ for user-to-item and item-to-user edges respectively. This essentially keeps only positive interaction edges, thereby reducing the cost of GNNs and absolving us of the responsibility of incorporating the rating into graph convolutions, which seem to have mostly been applied on unweighted graphs. However, it also misses out on potentially useful information in *how* positive an interaction is and the very existence of negative interactions. As a result, we mainly experimented with the former option.

Graph convolutions

When it comes to the graph convolutions to employ, there are many to consider. In Subsection 2.3.2, we covered two GNN architectures specialized for recommendation, NGCF [29] and LightGCN [10], both of which perform two very different graph convolution operations. NGCF is theoretically more expressive but very complex and computationally expensive, whilst LightGCN is simpler, much cheaper computationally and was even proven to perform better than NGCF in its authors' experiments.

It is important to note that both of these models were originally used for pure Collaborative Filtering on implicit interactions, while we seek to employ them for explicit interactions (i.e. ratings) on a weighted graph under a hybrid recommendation setting, where we do have access to node feature vectors (i.e. the fixed item and user profiles). As a result, picking the right convolution for our use case comes down to trial and error, as we seek to strike a balance between expressiveness (NGCF) and simplicity (LightGCN), as well as find a way to effectively incorporate the edges' weight into the graph convolutions.

After a lot of experimentation, we found that balance to be a modified LightGCN convolution, wherein we use a single learnable weight matrix W, instead of the original embeddings, during message construction. Despite the fact that the original LightGCN seemed to perform just as well[8], we believe that this learnable matrix is not too expensive

[8]Higher train MSE but similar validation MSE.

and it adds a lot of potential expressiveness in return, since we do have access to node features. Additional operations of NGCF like the element-wise multiplication or the second weight matrix of Equation 2.11 did not result in better performance, only in slower training times and sometimes worse performance, and, therefore, we omitted them.

Furthermore, in order to account for the heterogeneity[9] of the graph, we opted to use two of these matrices: one W_u for user-to-item edges and one W_i for item-to-user edges. Since the initial item and user vector distributions (i.e. the item profiles vs the user profiles) are different, it makes sense to allow our model to learn potentially different messages from each node type, instead of using one learnable layer for both. Note that this is no more computationally expensive than before, as we just use a different matrix depending on the type of edge and not both matrices at the same time as in NGCF. Furthermore, this allows us to use different user and item embedding dimensions, whereas they would have to be equal were we to use a single matrix for both node types.

Last but not least, in order to take into account the edge weights, we simply multiply each message with the edge's weight. Thus, if it's an edge with a negative weight (i.e. a negative interaction), the message will be the opposite of what it would be if it was a positive interaction.

All in all, we construct user-to-item and item-to-user messages at time step $t = 1, 2, \ldots, T$ as follows:

$$m_{u \to i}^{(t)} = \frac{r_{ui} - \overline{r_u}}{\sqrt{|N_u||N_i|}} \; W_u^{(t)} \, \vec{e_u}^{(t-1)} \tag{3.5}$$

$$m_{i \to u}^{(t)} = \frac{r_{ui} - \overline{r_i}}{\sqrt{|N_u||N_i|}} \; W_i^{(t)} \, \vec{e_i}^{(t-1)} \tag{3.6}$$

where $W_u^{(t)}$ and $W_i^{(t)}$ are the two learnable layers of the t-th GNN layer, $r_{ui} - \overline{r_u}$ and $r_{ui} - \overline{r_i}$ are edge weights as described earlier and $\vec{e_u}^{(t-1)}, \vec{e_i}^{(t-1)}$ are the user and item node embeddings of the previous GNN layer (or the initial ones when $t = 1$). In fact, we empirically found that using the same weight matrices W_u and W_i for all $t = 1, 2, \ldots, T$ (i.e. the exact same GNN convolution T times) is also viable in this use case, yielding equally good results.

For the neighbor aggregation step, we simply add all these (pre-normalized) messages at each receiver node as:

$$\vec{e_u}^{(t)} = \sum_{i \to u} m_{i \to u}^{(t)} \tag{3.7}$$

$$\vec{e_i}^{(t)} = \sum_{u \to i} m_{u \to i}^{(t)} \tag{3.8}$$

Similar to LightGCN [10], we do not include self-messages in favor of simply adding the initial user and item embeddings to the list of hidden states we aggregate to make up the final node embeddings. Just as in [10], this should make self-messages redundant and it did in fact empirically prove to be just as good if not better.

Finally, to get a final node embedding after all T GNN layers, we experimented with both: concatenating all the hidden states as in NGCF and taking the average of all the

[9]Item nodes are different from user nodes.

hidden state as in LightGCN. Compared to the former, the latter requires that all hidden states (including the original user and item embeddings) are of equal dimensions, but it is also cheaper and it worked equally well in our experiments.

Message and node dropout

In order to avoid overfitting the training graph, which seems to happen very easily when using GNNs for this task (especially if we use a lot of trainable weights like in NGCF), we experimented with the message node dropout techniques that the authors of NGCF suggested. For the former, during the training phase and on each forward pass, we randomly drop $p\%$ (e.g. 0.1, 0.2) of the *undirected* edges of the user-item bipartite graph before performing the T graph convolutions, so that the GNN layers learn not to rely on specific interactions. For the node dropout, we randomly drop all edges of $p\%$ of the nodes in the graph that are *not* in the current batch of user-item pairs, so that the GNN layers learn not to rely on specific nodes in the graph. Both techniques performed comparably and they did slightly improve the model's generalization ability, despite also slowing down convergence.

Caveat for target user-item pairs during training

Just like in Attention NCF, we also found it important to remove from the graph the edges concerning user-item interactions that we are trying to predict. That is, during training, we manually remove all edges concerning user-item pairs in the current batch, since the model will not have access to such edges when it gets called to predict new unknown user-item interactions during inference.

3.3 Prediction vs ranking problem

All models presented are designed to predict a preference score for a user-item pair. Hence, we can either train them to predict the preference scores we have access to (e.g. item ratings), thereby solving the prediction version of the recommendation problem (i.e. filling in the blanks of the utility matrix), or to predict an arbitrary preference score, with the sole intention that the predicted scores of items rank them well in relation to the user's ideal ranking(s), thereby solving the ranking version of the recommendation problem.

3.3.1 Solving the prediction problem

When we presented Neural Collaborative Filtering in Subsection 2.3.1.2, we showed how we can solve the prediction problem using pointwise losses like MSE (Equation 2.6) and BCE (Equation 2.7). Assuming we have access to ratings in the range of 0-5, it is more natural to opt for using the MSE loss, essentially training our models for a regression task. That being said, we also experimented with using BCE loss for soft targets (aka BCE loss with logits) by squashing 0-5 ratings to [0, 1], representing the probability of really liking an item. Since the results between the two losses were very similar, we preferred the MSE loss due to its better interpretation.

To solve the prediction problem, we simply treat each training interaction between a user u and an item i with a rating r as a triplet (u, i, r), where r is the target label we are trying to predict.

3.3.2 Solving the ranking problem

To solve the ranking problem, we employed the BPR loss (Equation 2.8) that was presented in Subsection 2.3.1.2. This pairwise loss is meant to be used on (*user, item, item*) triplets (u, i, j) where item i should be ranked higher than item j for user u. Assuming we have access to ratings r in the range of 0-5 for each user-item interaction, we can, for each user u, form triplets (u, i, j) where $r_{ui} > r_{uj}$.

The problem is that all these possible triplets are vastly more in number than the original known interactions. Going through all of them in an epoch would take prohibitively long. To combat this, we fix all the possible (u, i) pairs before training, thereby keeping the amount of training samples seen in an epoch equivalent to that of pointwise learning, and, during training, for each sample (u, i) we randomly sample the item j from all its possible options. Statistically, a different item j will be sampled in each epoch for the same (u, i) pair. Note that, out of the many possible options for j, only one will be considered per (u, i) pair in an epoch. As a counter measure, we opted to use bigger batch sizes and/or smaller learning rate in this case so that we would end up completing more epochs before eventually overfitting, thereby getting to see more triplets. All in all, sampling just one item j proved to be enough to get decent results that were comparable to pointwise learning, judging by the ranking quality of unseen interactions.

Of course, there are many ways to perform this negative sampling of j. Perhaps the simplest thing we can do is to sample j uniformly among all its options. This may however have the undesired effect of suffering from slow convergence as a result of vanishing gradients as discussed in [22], since we may sample items that were already ranked correctly. Given that incorrectly ranked pairs are the most informative, another strategy that may make more sense is to give certain options for j higher probability of beings sampled based on how *hard* we *expect*[10] it to be for them to be ranked correctly. These items are typically referred to as *hard negatives*. In our case, where we have access to 0-5 ratings, one could argue that the higher the rating of item j by user u then the harder we can expect it to be for a model to correctly rank that item below item i. Therefore, we could give higher probability to candidate j items the higher their rating by user u is, in the hopes that this will help us sample more informative negatives.

The most intuitive and efficient way that we came up with to do that is to assign probabilities to all possible negatives with the following formula:

$$p_j = \frac{r_{uj}^{w}}{\sum_{k \in Negatives(u,i)} r_{uk}^{w}} \quad \text{for each } j \in Negatives(u, i) \tag{3.9}$$

[10]We can't know before we actually make the prediction and it's very unlikely we will sample the same j twice to even bother with something like boosting.

where *Negatives*(u, i) are all possible options for j and $w \in [0, \infty)$ is a hyperparameter that controls how much more weight we should assign to higher rated negatives compared to lower rated ones. For $w = 0$ we end up with uniform sampling, while increasing it boosts higher rated items more. Regardless of the value of w, we always end up with $\sum_j p_j = 1$. We also tried using softmax, but that is a much more expensive operation for us to pay every time we need to sample a single negative.

In our experiments, we found that we should not overdo it with w (e.g. using only values in $0 - 3$), as we don't want to *only* sample the hardest of negatives, since we still need the network to know how to rank items with low ratings as well. The method that we ended up using is to gradually increase w, starting from 0 (uniform sampling), as we complete more epochs and the model has learned more, essentially making the training problem harder as we go.

Chapter 4

Experiments

4.1 Dataset

In order to train and evaluate the neural network models presented in the previous chapter, we created a custom dataset of user-item interactions as well as item features for each item.

4.1.1 Public datasets used

To create this custom dataset, we relied on the following two public datasets.

MovieLens 25M dataset

The MovieLens Dataset[1] is one of the most popular Collaborative Filtering datasets out there, used as a benchmark for mainly pure Collaborative Filtering methods. In its latest stable release (published in December 2019), it contains 25 million movie ratings from 160000 users to 60000 movies.

In addition to user ratings of movies, this dataset also contains around 1100 tags, which they call *genome tags*, describing the topics of the movie (e.g. addiction, World War II, 18th century, etc.). For each movie and tag pair, there is a predicted relevance score in $[0, 1]$, which describes how relevant this tag is for the movie. In this work, these tags were used as additional item features.

IMDb dataset

The IMDb dataset[2] is the official dataset provided by IMDb, which is updated regularly. It contains metadata for movies such as their genres, directors, actors, writers, etc. Luckily, the MovieLens dataset includes the IMDb ID for each movie, which makes it possible to combine the two datasets. We use this metadata, along with the genome tags from MovieLens, in order to build item profiles for the movies, as we described in Subsection 3.1.1[3].

[1]https://grouplens.org/datasets/movielens/
[2]https://www.imdb.com/interfaces/
[3]Here all features happen to be categorical.

4.1.2 Creating a custom dataset

The deep learning models described in the previous chapter are increasingly expensive, in both time and space. As a result, one cannot possibly use all the 25 million ratings in the MovieLens dataset for benchmarking without having access to considerable computing resources and/or significant amounts of time. Therefore, we selected only a subset of the original movies and users and considered all the interactions available for them.

To pick a subset of movies to focus on, we apply the following filters:

1. The movie must have an IMDb entry in order to get metadata for it.

2. The movie must be in a specific time frame. In this way, we focus on more recent movies, e.g. from 2000 and later.

3. The movie must have a minimum number of ratings from the original 160000 users and a minimum number of votes in the IMDb website. This way, we focus on movies that are at least somewhat popular and are bound to have more user interactions than unpopular ones[4].

After adjusting these parameters to get an acceptable number of items, we pick a subset of all the users. First, we filter out users who have less than 100 or more than 400 interactions. Then, if we are still left with more users than desired, we randomly sample the exact number of users that we want to use.

By applying the above process we settled on a dataset of 910891 interactions (i.e. ratings) between 1174 movies and 5000 randomly selected users. Some metadata for those movies are shown in Figure 4.1. The rating distribution for items and the degree distributions for both items and users compared to the original MovieLens data are shown in Figure 4.2. Notice that both the item and user degree distributions roughly follow a Power-Law distribution, as is usually the case in practice.

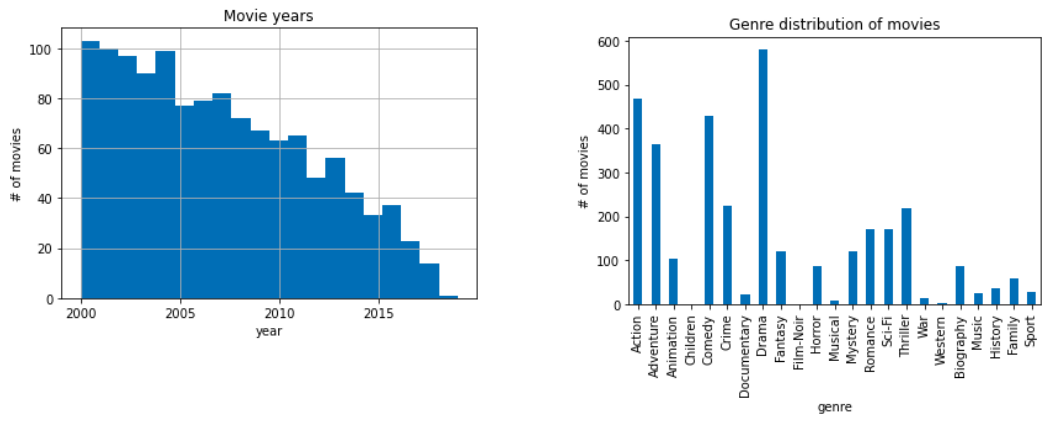
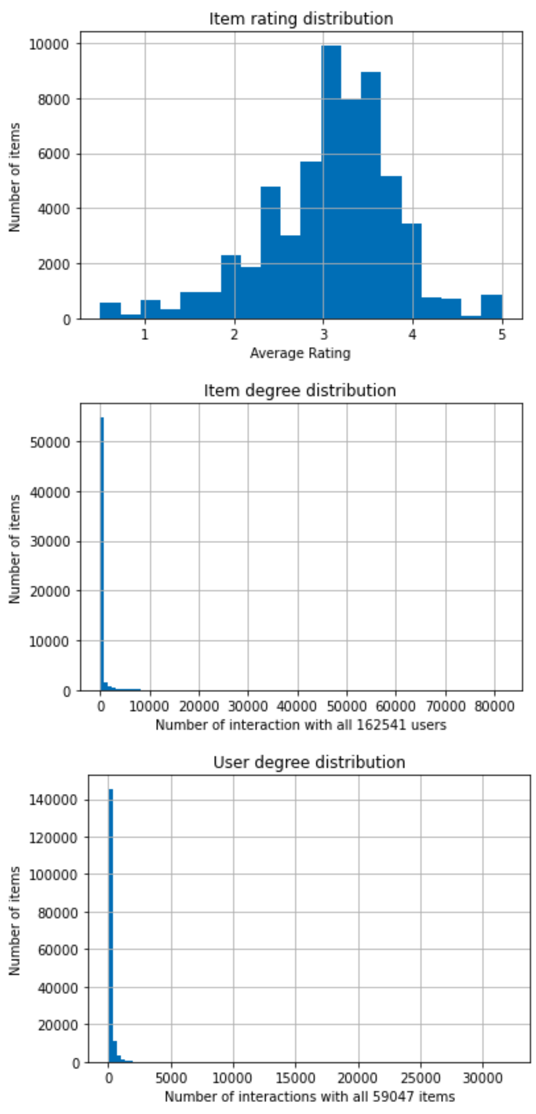
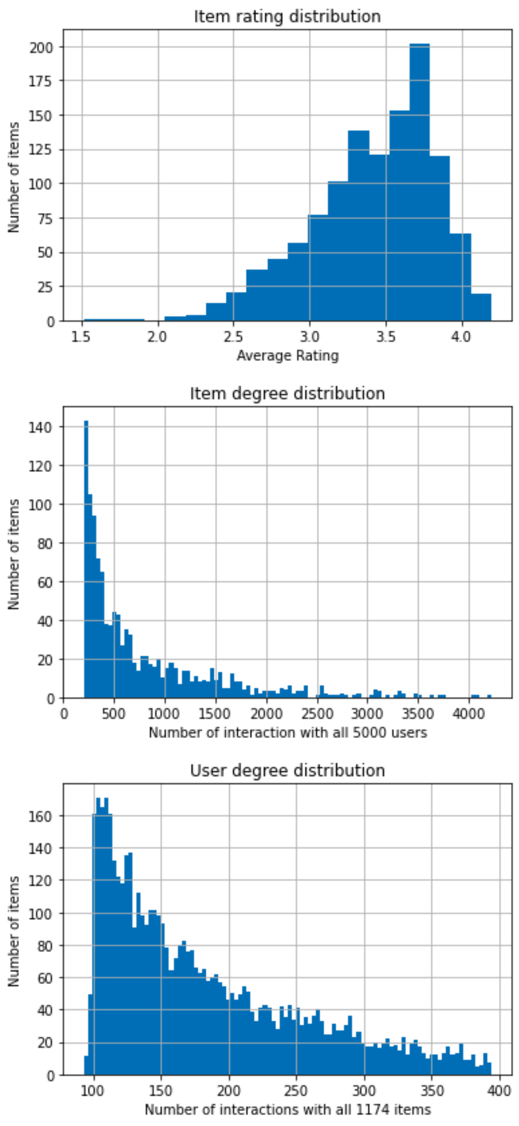


Figure 4.1. *Movie year and genre distribution.*

[4]Recall that Collaborative Filtering methods are known to struggle when faced with too much sparsity.



(a) *In the original movieLens dataset.*



(b) *In our custom dataset.*

Figure 4.2. *The item rating distribution and the item and user degree distributions.*

4.1.3 Train-val-test split

For point-wise learning, we train our model on user-item interaction pairs. The fact that we have a user dimension and an item dimension, however, makes it unclear what the best strategy for splitting the interaction data in train, validation and test sets may be. A variety of possible choices are presented in [19]. We note the following:

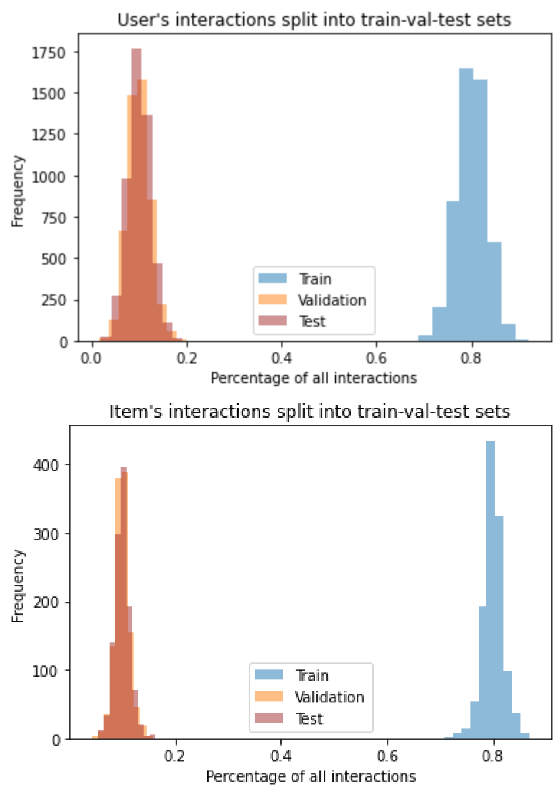
1. The simplest and most straightforward strategy is to randomly and at uniform split all the interactions into train, validation and test sets (without considering which item or user any of them correspond to). Due to the uniformity in sampling, users and items with more interactions are expected to be just as more common in the validation and test sets as they are in the train set.

2. Another strategy that is very common [19] is to randomly leave out one or another fixed number of item interactions per user for the test and validation sets and use the

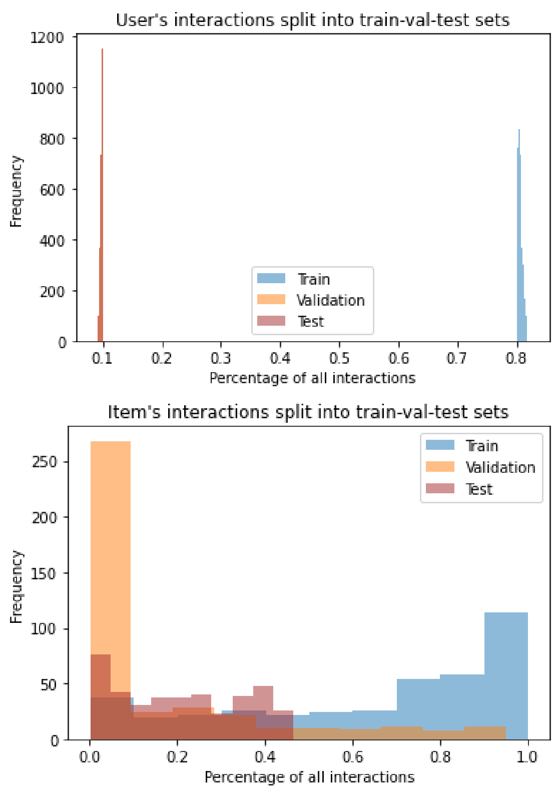
rest for training. This strategy has the property that each user is equally represented in the validation and test sets, despite him having more or less interactions than other users overall. We could also do this from the item's perspective, although the user's one appears to be much more common. We could also account for the difference in number of user ratings by leaving out a certain percentage (e.g. 5%, 10%) of all their interactions instead of a fixed number, thereby giving higher importance to users we know more about.

3. Some apply the above strategy, but instead of leaving out random entries, they leave out the latest-in-time ratings (assuming we have a timestamp for each rating). This strategy is called temporal splitting and it appears to be more appropriate in cases where we consider the temporal aspect of user-item interactions (e.g. models who treat them as a temporal sequence).

4. Another sensible strategy is to split the interactions based on the users, meaning that a user along with all of his interactions are placed on the same set. This aims to better measure the model's ability to generalize to completely new users. This strategy, however, is only applicable to models that are in fact able to generalize to new users (e.g. we cannot do this when using one-hot vectors for users).



(a) *Using the first flat method of splitting interactions.*



(b) *Using the second popular method of splitting interactions based on users.*

Figure 4.3. *Histograms of how the user-item interactions are distributed between train, validation and test sets.*

From the strategies mentioned above, we mostly experimented with the first and the second approach. Figure 4.3 shows how these methods distribute user-item interactions among the train, validation and test sets as grouped by users and by items. We can see that, by using the first method, each item and each user has roughly (within reasonable variance) 80% of its interactions as training interactions, 10% as validation and 10% as test interactions, which is exactly what we would want for Collaborative Filtering purposes. By using the second method, each user's interactions are almost exactly split at 80-10-10, but at the same time the item's interactions are distributed more chaotically, with some items having all their interactions in the train set, some having as much as 80% or 90% of their interactions at the validation set, etc. All in all, judging by Figure 4.3 and by running some experiments using both splitting methods[5], we decided to opt for the first splitting method on account of its simplicity and how much better it distributes both the users' and the items' interactions.

A couple of caveats to note about the splitting of user-item interactions are:

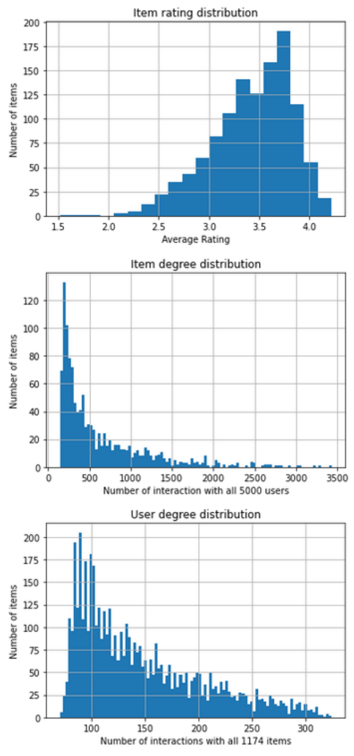
- For the user profile construction we, of course, only use interactions that are in the train set, as we have to consider the ones in the validation and test sets as unknown interactions that we do not have access to for training. That being said, there is an argument to be made about including the validation interactions in the user profile construction when we are evaluating our trained models on the test set. The idea is to see if the models would perform better by knowing more about the users, despite not having been trained on those validation interactions, or if they would perform worse[6].

- The more interactions we keep in the train set, the better our models appear to do when we evaluate them on the validation and test sets. This is probably due to them having been trained on more data (less sparsity in the utility matrix) and the fact that user profiles are more accurate (by using more interactions). At the same time, however, we want to use enough interactions for validation and testing so that our evaluations on them are statistically significant. We found a good compromise to be to keep 80% of all interactions of training, 10% for validation and 10% for testing.

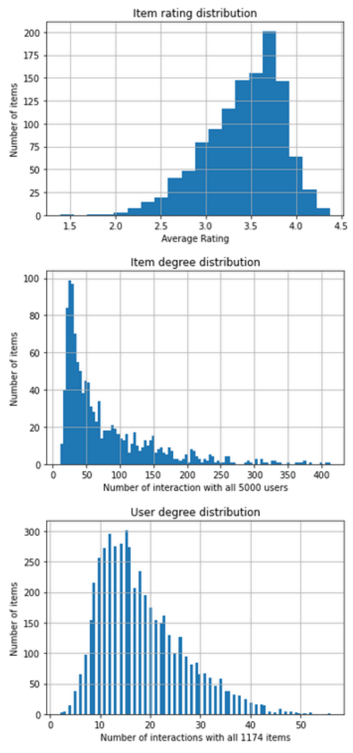
After performing the split we are left with the statistics shown in Figure 4.4 for our train, validation and test sets respectively. Notice how closely they resemble the statistics of each other as well as the original dataset's in Figure 4.2.

[5]The models faired roughly the same compared to each other in both experiments except that by using the second method the one-hot models performed much worse (which is reasonable considering there were items at the test/val set which were not in the train set) and all the metrics were somewhat worse.

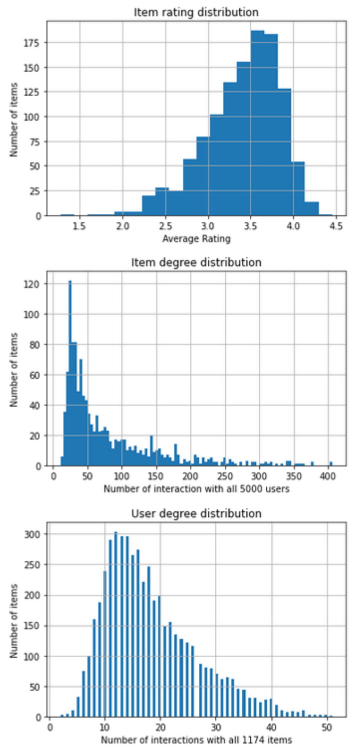
[6]For example, because they have overfitted on the training user profiles



(a) *In the train set.*



(b) *In the validation set.*



(c) *In the test set.*

Figure 4.4. *The item rating distribution and the item and user degree distributions.*

4.2 Evaluation metrics

4.2.1 Regression metrics

In order to evaluate the models trained for regression on the prediction version of the recommendation problem we can use the MSE loss (Equation 2.6) or, its square root, the RMSE on the corresponding dataset (e.g. the validation and test sets). These regression metrics give us an idea of how close to the correct ratings our model's predictions lie. In fact, the RMSE is exactly the standard deviation of the prediction error. Furthermore, we can plot a histogram of the predicted scores compared to their ground truth, in order to help us visualize how well the model is doing.

4.2.2 Ranking metrics

When it comes to the models trained for the ranking version of the recommendation problem using the BPR loss, we can not use the MSE or the RMSE, as we are not performing regression and the predicted scores are not predicted ratings. They are just arbitrary numbers, which can only be judged in relation to each other. Therefore, we need to evaluate them based on the ranking quality of items, that they produce for users after ordering the items based on their predicted scores. To measure this ranking quality of items, we need a way to measure how close each item ranking is to an ideal item ranking, i.e. the ranking (or one of the rankings in case of ties) that we would get if we were to order items based on their actual ratings.

A popular ranking measure for this task is the Normalized Discounted Cumulative Gain (NDCG). This metric takes as input the actual ratings and the predicted rating for each item that a user has rated and returns a ranking score. The higher this score, the better the ranking is. To define NDCG, we first have to define its simpler predecessors.

For a single user, i.e. for a sequence of items ranked by their predicted relevance scores in descending order, the Cumulative Gain (CG) at cutoff k is defined as:

$$CG@k = \sum_{i=1}^{k} relevance_i \tag{4.1}$$

where the cutoff k (e.g. usually 5, 10 or 20 items) defines how many of the top recommendations we care about and $relevance_i$ is the ground truth for how relevant an item is. In this case, we can just use its actual rating by the user in question as its relevance.

However, when using the Cumulative Gain, it does not matter how well ranked the top k items are, it only matters that the most relevant items made it to the top k. In order to also differentiate between different orderings of the top k recommended items we can use the Discounted Cumulative Gain (DCG) at cutoff k which is defined as:

$$DCG@k = \sum_{i=1}^{k} \frac{relevance_i}{log_2(i+1)} \tag{4.2}$$

It's called *discounted* because we discount each relevance score by an increasingly higher number, as given by the term $log_2(i+1)$, thereby making earlier ranked relevance scores more important than later ranked ones. That is, we achieve the best possible DCG by ordering the predicted scores in decreasing order.

Even so, there is still a problem with DCG. As different users may have different relevance scores or some may have less than k recommendations, the scale of their DCGs may be different. To combat this, the Normalized Discounted Cumulative Gain (NDCG) normalizes the DCG score of each user (i.e. of each ranked sequence) in [0, 1] as:

$$NDCG@k = \frac{DCG@k}{iDCG@k} \tag{4.3}$$

where iDCG@k is the ideal DCG that we get by using the actual relevance scores as the predicted ones. This normalization makes the NDCG of each user be at the same scale. By taking the mean of the NDCG@k of all the users, we are left with a final NDCG metric by which to judge our model in terms of ranking quality.

Note that, even though the NDCG is normalized in [0, 1], if all the relevance scores are only positives (as is usually the case) then in all likelihood the NDCG will never actually be zero or even be close to it, even for the worst possible ranking that we could produce (e.g. increasing order of actual ratings). If we want its values to be better distributed in [0, 1], where 1 represents the ideal ranking and 0 represents the worst possible ranking, then we could use a min-max normalization scheme as:

$$adj\text{-}NDCG@k = \frac{DCG@k - wDCG@k}{iDCG@k - wDCG@k} \tag{4.4}$$

where wDCG@k is the worst DCG, that we get, for example, by using the opposite of the actual relevance scores as the predicted ones. Both NDCG and adj-NDCG behave the exact same way and they will help us pick the same best model. The only difference is that, in our opinion, adj-NDCG has a more intuitive interpretation of its face value.

4.3 Experiments

Having split the data into train, validation and test sets, we proceeded to train and evaluate the models we discussed in the previous chapter on the custom dataset we described in Subsection 4.1. In this Subsection, first, we will focus on each model type separately and show experiments with different options for each type's hyperparameters. Then, we will compare the best model of each type with each other and look at the full picture, as well as answer interesting questions that arise from this work.

While the validation set is used for early stopping during training, the test set is used to produce a final evaluation of the model. During this evaluation we can keep the original user profiles for each user, which use only known user-item interactions from the train set, or we can add to them the user-item interactions of the validation set[7]. Doing the latter should theoretically produce more accurate user profiles and, thus, improve the test metrics, despite the fact that the model has not been trained on those validation interactions and more accurate user profiles[8]. To that end, we calculate the test metrics using both methods and we distinguish the latter by dubbing these metrics as *Test+*, to mark the addition of the validation interactions in the input.

4.3.1 Hyperparameter exploration

All the three architectures, Basic NCF, Attention NCF and Graph NCF, share some common hyperparameters (e.g. those of the MLP that makes the final prediction), whilst also having unique ones that are specific to their architecture. Before comparing each model type with each other, it is important to consider different hyperparameter options for each one individually.

Basic NCF

Basic NCF is the simplest architecture of the three and, as such, doesn't have a lot of significant hyperparameter options to consider. For Basic NCF, we explored different sizes for the user and item embedding layers (we used the same size for both), the possible inclusion of an activation function (e.g. ReLU) at those layers as well as different hidden layers for the final MLP network (which do have a ReLU activation function). The final test metrics for some configurations are shown in Table 4.1. For all these runs, after some initial tuning, we used a learning rate of 0.0007, a batch size of 128, a dropout rate of 0.2 and a weight decay of 0.00001 (L2 regularization).

[7]Note that for Graph NCF we also add these validation interactions as edges in the user-item bipartite graph.

[8]If not, then it is possible that we have overfitted to the user profiles of the train set.

Table 4.1. *Test set evaluation for different Basic NCF hyperparameters.*

Emb dim	Emb activation	MLP hidden layers	Test MSE	Test+ MSE	Test NDCG@10	Test+ NDCG@10
128	ReLU	128	0.5694	0.5670	0.9308	0.9313
64	–	128	0.5678	0.5651	0.9313	0.9319
128	–	128	0.5681	0.5654	0.9318	0.9324
128	–	256	**0.5657**	0.5618	0.9320	0.9327
256	–	256	0.5666	0.5632	0.9311	0.9317
256	–	512	0.5700	0.5665	0.9308	0.9318
128	–	128, 64	0.5716	0.5698	0.9320	0.9324
128	–	256, 128	0.5708	0.5680	0.9318	0.9325
128	–	256, 128, 64	0.5779	0.5757	0.9312	0.9314
256	–	512, 256	**0.5657**	**0.5614**	**0.9329**	**0.9335**
256	–	512, 256, 128	0.5725	0.5702	0.9321	0.9329

As we can see, changing these layer sizes has little effect on the final test metrics, possibly thanks to the regularization methods employed (dropout, L2 normalization). That being said, we picked the cheapest model with the best test MSE of 0.5657 as a representative model for later comparisons.

Attention NCF

When it comes to Attention NCF, other than the layer sizes that are in common with Basic NCF, we can also experiment with message dropout, the layer sizes of the AttentionNet network and with using cosine similarity instead of the AttentionNet altogether, as we described in Subsection 3.2.2. For the AttentionNet, we mostly experimented with a network with one hidden layer, whose size is a hyperparameter, or one with no hidden layers. The test metrics for some of these configurations are shown in Table 4.2. For all these runs, we used a learning rate of $0.0007 - 0.001$, a batch size of 512, a dropout rate of 0.2, a weight decay of 0.00001 (L2 regularization) and an item and user embedding dimension of 128.

Table 4.2. *Test set evaluation for different Attention NCF hyperparameters.*

Method	AttentionNet dim	MLP hidden layers	Message dropout	Test MSE	Test+ MSE	Test NDCG@10	Test+ NDCG@10
Cosine Similarity	–	256, 128	0.0	0.5420	0.5394	0.9360	0.9364
AttentionNet	–	256, 128	0.0	0.5666	0.5645	0.9329	0.9332
AttentionNet	128	256, 128	0.0	**0.5244**	**0.5185**	0.9387	0.9396
AttentionNet	256	256, 128	0.0	0.5255	0.5190	**0.9388**	**0.9400**
AttentionNet	128	256, 128	0.1	0.5277	0.5237	**0.9388**	0.9397
AttentionNet	128	256, 128	0.2	0.5311	0.5278	0.9376	0.9381
AttentionNet	128	256	0.0	0.5315	0.5258	0.9371	0.9384

From the experiments above, it is clear that Attention NCF performs better with the AttentionNet method when the AttentionNet network has at least one hidden layer of sufficient size, while it performs much worse when it has no hidden layer. Furthermore, it is clear that the AttentionNet method is superior to the Cosine Similarity method, possibly because the latter is naturally symmetric as an attention mechanism, while the former is not, allowing for a lower bias model. Message dropout did not seem to help much after all. Out of these runs, we chose the model with the lowest Test MSE of 0.5244 as the representative of this architecture.

Graph NCF

Graph NCF is the most complicated architecture we tackled and, thus, there are a lot of different options for many parts of it, e.g. different ways to model the user-item bipartite graph, many different graph convolution operations, different ways to combine the hidden states into one node embedding, etc. It took a lot of trial and error to settle on the final architecture that we describe in Subsection 3.2.3. However, even this version has some hyperparameters to test on our final dataset.

In Table 4.3 we explore the use of different sizes and number of GNN layers, different message and/or node dropout rates and different ways to aggregate the $T + 1$ hidden states (including the original user and item embeddings) into one final node embedding and different MLP networks or the use of dot product instead of an MLP as in [29, 10]. For all these runs, we used a learning rate of 0.001, a batch size of 512, a dropout rate of 0.2 and a weight decay of 0.00001 (L2 regularization). The number of GNN layers shown does not include the initial user and item embeddings layers, which have both the same dimension as the GNN layers. Furthermore, all the GNN layers share the same weights, as this was found to be empirically better.

Table 4.3. *Test set evaluation for different Graph NCF hyperparameters.*

GNN layers	MLP hidden layers	Message / Node dropout	Final aggregation	Test MSE	Test+ MSE	Test NDCG@10	Test+ NDCG@10
1 × 128	128	0.2 / 0.0	mean	0.5711	0.5683	0.9309	0.9316
2 × 128	128	0.1 / 0.0	mean	0.5684	0.5648	0.9312	0.9320
2 × 128	128	0.2 / 0.0	mean	0.5670	0.5646	0.9318	0.9320
2 × 128	256, 128	0.2 / 0.0	mean	**0.5667**	**0.5632**	**0.9321**	0.9324
3 × 128	128	0.2 / 0.0	mean	0.5675	0.5645	0.9312	0.9315
1 × 64	128	0.2 / 0.0	mean	0.5694	0.5659	0.9316	**0.9327**
2 × 64	128	0.2 / 0.0	mean	0.5732	0.5704	0.9305	0.9312
3 × 64	128	0.0 / 0.0	mean	0.5796	0.5770	0.9299	0.9304
3 × 64	128	0.2 / 0.0	mean	0.5742	0.5713	0.9304	0.9307
3 × 64	128	0.0 / 0.2	mean	0.5753	0.5727	0.9297	0.9303
3 × 64	128	0.1 / 0.1	mean	0.5746	0.5721	0.9305	0.9310
3 × 64	128, 64	0.2 / 0.0	mean	0.5711	0.5683	0.9319	0.9324
4 × 64	128	0.2 / 0.0	mean	0.5687	0.5664	0.9312	0.9318
2 × 64	128	0.1 / 0.0	concat	0.5750	0.5728	0.9302	0.9306
3 × 64	128	0.1 / 0.0	concat	0.5708	0.5685	0.9314	0.9318
3 × 64	dot product	0.0 / 0.0	mean	0.5867	0.5807	0.9299	0.9314
3 × 64	dot product	0.1 / 0.1	mean	0.5762	0.5717	0.9316	0.9324

As we can see, all these different configurations made little difference to the final test metrics. The most notable difference is observed for the models where we use the dot product instead of the MLP to make the final prediction, as these models achieve significantly smaller train MSE (not shown in the table) but slightly higher test MSE. Message and node dropout, even when used together, only slightly improve the generalization of the model but, at the same time, increase the number of epochs needed to converge. The number of GNN layers and the final node embedding aggregation method also do not seem to make much of a difference. Last but not least, it is worth reiterating that, other than the experiments shown in Table 4.3, we have also extensively tried a lot of different GNN architectures before settling on this one and they all performed similarly or worse in the test set. For the purposes of further comparisons, we again chose the model with the smallest test MSE of 0.5667 as the representative model.

4.3.2 Using one-hot vectors vs using features

Given our work on using content-based user and item profiles, one important question to ask ourselves is how much of a role do they play in the performance of our models. We know that without them, it would be impossible to generalize to new items and/or new users. But it would be interesting to know if they also result in better recommendations compared to using plain one-hot vectors as in pure CF.

To put that to the test, we took our simplest model, Basic NCF, and trained it using the best hyperparameters from Subsection 4.3.1 and three different input configurations. Once using one-hot vectors for both users and items, once using features for items but one-hot vectors for users, and once using both item and user profiles as inputs. The learning curves for these experiments are shown in Figure 4.5 and their test metrics are shown in Table 4.4.

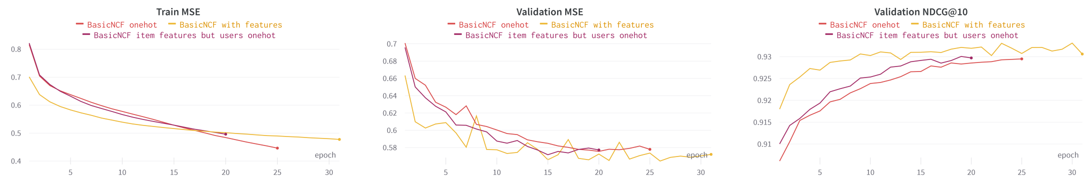


Figure 4.5. *Learning curves for Basic NCF using one-hot input vectors vs using feature vectors.*

Table 4.4. *Test set evaluation for Basic NCF using one-hot input vectors vs feature vectors.*

Model	Test MSE	Test+ MSE	Test NDCG@10 / adjusted	Test+ NDCG@10 / adjusted
Basic NCF one-hot	0.5749	–	0.9287 / 0.7712	–
Basic NCF with features but users one-hot	0.5739	–	0.9287 / 0.7706	–
Basic NCF with features	**0.5657**	0.5618	**0.9320** / **0.7786**	0.9327 / 0.7817

As we can see by the results, it appears that adding item features does somewhat help our model to generalize better, as that respective model showcases slightly smaller validation and test MSE as well as bigger validation and test NDCG@10. Whilst, at first glance, adding user profiles does not appear to make much of a difference in terms of actual performance, we need to keep in mind that the more accurate those user profiles become, the better the performance we can expect, as is evident from the *Test+* metrics (note that, by definition, these metrics cannot be employed when we are using one-hot vectors for users).

By also performing a similar one-hot vs features experiment for the Graph NCF model, we notice the same pattern, as shown in Figure 4.6 and Table 4.6. It is therefore safe to conclude that we can introduce more generalizable patterns for our neural network models to learn by using content-based profiles. However, the improvement or lack there of one will definitely depend on the quality of those content-based profiles in relation to the task at hand.

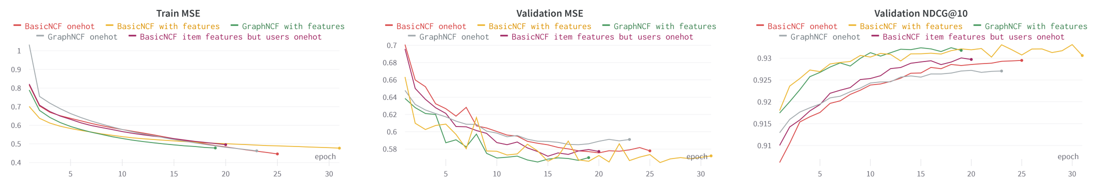


Figure 4.6. *Learning curves for Basic NCF and Graph NCF using one-hot input vectors vs using feature vectors.*

4.3.3 Comparing all models

Another question to ask ourselves is how does each model type perform compared to each other. In general, Attention NCF will be more computationally expensive than Basic NCF and, in turn, Graph NCF will be more computationally expensive than Attention NCF. Of course, the actual training (and inference) time will depend on the layer sizes, the batch size, the time spent in the data loader and many other factors. Nevertheless, a rough estimate of minutes per epoch that it took for each model type in our experiments by using a CUDA GPU and having optimized the data loading process (e.g. custom collate, indexing, etc.) is presented in Table 4.5 and it does verify this intuition.

Table 4.5. *Average time per epoch for each model type.*

Model	batch size	Average time per epoch (minutes / epoch)
Basic NCF	128	2.516
Basic NCF	512	2.037
Attention NCF with AttentionNet	512	5.281
Attention NCF with Cosine Similarity	512	5.454
Graph NCF with 3×64 GNN layers	512	5.714
Graph NCF with 2×128 GNN layers	512	6.842

Nevertheless, that does not necessarily mean that the more expensive and complicated models will also perform better in our dataset. After a lot of experimentation, the best run (in terms of test set metrics) for each model type is presented in Figure 4.7 and its test metrics in Table 4.6.

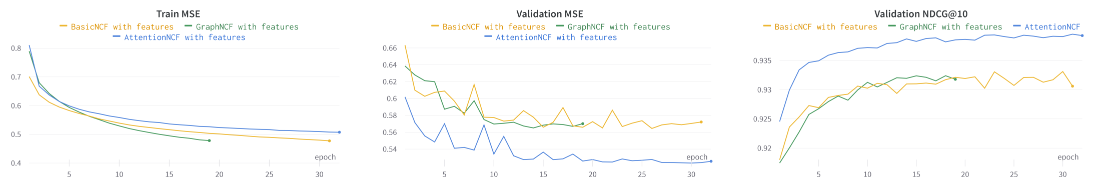


Figure 4.7. *Learning curves for all models using features.*

Table 4.6. *Test set evaluation for all models.*

Model	Test MSE	Test+ MSE	Test NDCG@10 / adjusted	Test+ NDCG@10 / adjusted
Basic NCF	0.5657	0.5618	0.9320 / 0.7786	0.9327 / 0.7817
Attention NCF	**0.5244**	**0.5185**	**0.9387 / 0.8009**	**0.9396 / 0.8039**
Graph NCF	0.5667	0.5632	0.9321 / 0.7804	0.9324 / 0.7817
Basic NCF one-hot	0.5749	–	0.9287 / 0.7712	–
Graph NCF one-hot	0.5858	–	0.9277 / 0.7690	–

As we can see, Attention NCF is performing significantly better on the validation and test sets compared to Basic NCF and Graph NCF, meaning that it is better at generalizing. More specifically, it is 7.3% better at Test MSE and 7.7% better at Test+ MSE from Basic NCF. Notice that its training loss is also bigger than that of Basic NCF and Graph NCF. We believe that, other than the item-item attention mechanism (which is also important), a definite factor contributing to both these facts is that, in Attention NCF, we are able to mask out the target training user-item interaction from being used in the user profile construction. On the contrary, we are naturally unable to do so in Basic NCF and Graph NCF where we rely on fixed user profiles, as explained in Subsection 3.2.2. This makes the training task of Basic NCF and Graph NCF a little easier than the task of inference, because we have access to the target user-item interaction as a *small* part of the user profile during training but not during inference.

Surprisingly, despite being more computationally expensive and potentially more expressive, Graph NCF appears to perform no better than Basic NCF. This contradicts our initial expectations, but then again, perhaps the collaborative signal on the user-item bipartite graph, that we are trying to explicitly capture with graph convolutions in Graph NCF, is not strong enough to offer any advantage over implicitly capturing it with Basic NCF. After many experiments with many different GNN architectures, no GNN architecture managed to outperform Basic NCF. Further experiments with even simpler methods than Basic NCF also support this intuition, as we shall see in Subsection 4.3.7.

Of course, all of these models are better than their one-hot counterparts, as shown in Figure 4.8 and Table 4.6.

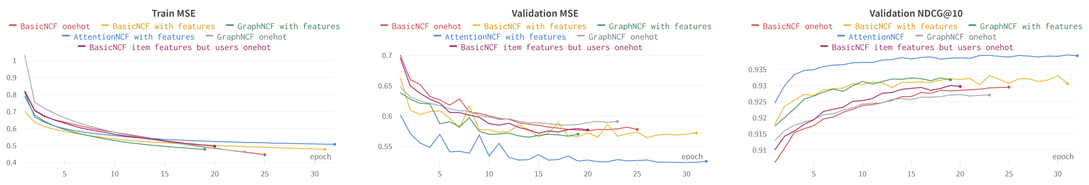
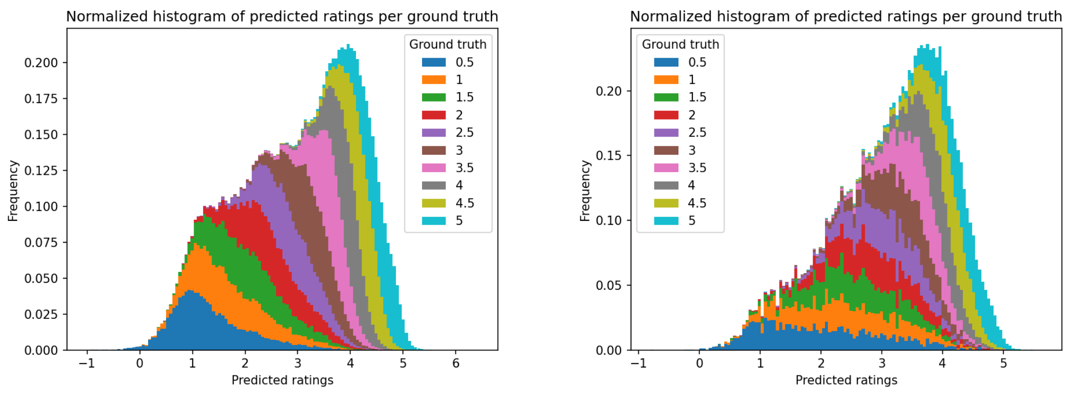


Figure 4.8. *Learning curves for all models.*

4.3.4 Visualizing the quality of the regression

To get a visual idea of how well the regression is working, we took our best Attention NCF model and plotted a number of stacked histograms – one per ground truth rating – of all the predicted ratings in the train and test sets, regardless of which user-item pair they belong to. Since the distribution of the ground truth ratings, as shown in Figure 4.2b, is not uniform (e.g. there are very little 1-2s compared to 3-4s), we normalized all the stacked histograms so that they take up an equal amount of space to allow for an easier interpretation. These normalized stacked histograms are shown in Figure 4.9.

Notice that, even though a lot of predicted ratings are off (especially for rare ground truth ratings e.g. 0.5 to 1.5), there definitely exists a noticeable distinction between the predicted scores of different ground truths, particularly in the train set. This means that the model has learned, to some extent, to correctly rank a lot of the user-item interactions.



(a) *In the train set.* **(b)** *In the test set.*

Figure 4.9. *Normalized stacked histograms of predicted ratings per ground truth ratings.*

4.3.5 Visualizing the item-item attention in Attention NCF

A considerable advantage that Attention NCF holds over Basic NCF and Graph NCF is the explainability it offers through the attention weights it calculates, when dynamically creating a user profile from rated item profiles as in Equation 3.4. Recall that, in order to make a prediction for a user-item interaction between a user u and a candidate item c, Attention NCF calculates an attention weight a_{ci} between item c and every rated item i by user u. By looking at these attention weights during a prediction, we can observe which of the user's rated items the model gives more attention to, in order to make said prediction. In other words, when offering a user a recommendation of a candidate item we can also supply him with a list of items that he has rated before and have high attention scores, as an explanation of why this candidate item is being recommended to him.

In the direction of visualizing this item-item attention mechanism, we run inference on the entire test set and aggregated some statistics for the attention weights that were calculated during the whole process. More specifically, for each candidate item and rated item pair we calculate the average attention weight the pair received, as in the sum of the attention weights that were calculated for that pair divided by the count that the pair was observed. This average attention weight may not exactly be the fairest of measures, since we are not accounting for the fact that the attention is distributed over the different sets of rated items via softmax[9]. Nevertheless, we believe that it is a good high-level indication of how the attention mechanism is working overall, independently of a specific user with a specific set of rated items, in the unseen user-item interactions of the test set.

We visualize these average attention weights for two arbitrarily chosen subsets of all the items in the heat maps of Figure 4.10. As one can see, in the first heat map the model has learned to generally give more attention to item-item pairs of movies from the same franchise e.g. the Lord of the Rings, Spider-Man, Harry Potter, etc., which makes sense as how one has rated one of those movies may tell a lot about how he is going to rate the other ones. In the second heat map we can also notice how it has learned to give higher

[9]Different users are associated with a different number of rated items. The more rated items exist, the less attention each one of them would be expected to receive, all else being equal

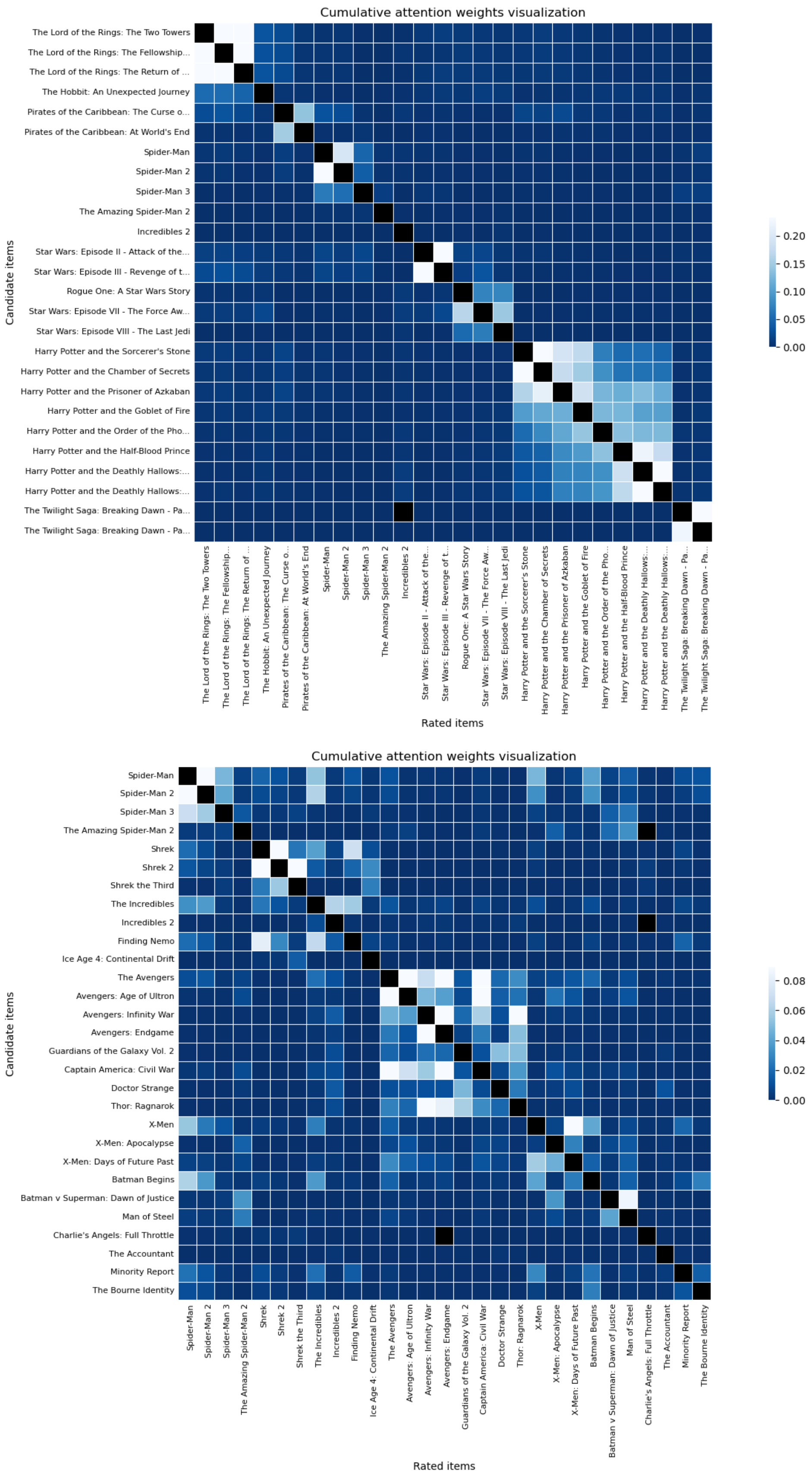


Figure 4.10. *Average item-item attention weights visualized for two subsets of movies.*

attention to movies pairs that are not from the exact same franchise but are similar in nature e.g. cartoons, superhero movies, etc.

Keep in mind that these attention weights are learned solely from the item profiles of the two movies at hand. Since we used the AttentionNet method that we described in 3.2.2, these item-item attention weights between candidate and rated items are not symmetric, as one can clearly see from Figure 4.10. If we were to use the cosine similarity method, then these attention weights would be symmetric. However, as shown earlier, the AttentionNet method yielded better results.

4.3.6 Solving the ranking problem directly

As we discussed in Subsection 3.3.2, by employing the BPR loss on (u, i, j) triplets of (*user, item, item*), we can train the same three types of models for the ranking version of the recommendation problem directly. In this case, we use the BPR loss only as a training loss and not as a validation loss, as we deemed it more suitable to use the validation NDCG metric as the early stopping criterion. Getting better at the ranking problem should directly translate to better NDCG values, so it makes sense to pick the model with the best validation NDCG.

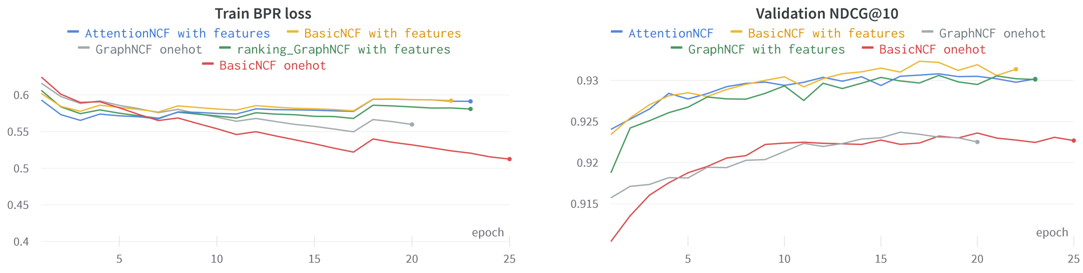


Figure 4.11. *Learning curves for all models when training for ranking with BPR loss.*

Table 4.7. *Test set evaluation for all ranking models.*

Model	Test NDCG@10 / adjusted	Test+ NDCG@10 / adjusted
Basic NCF	**0.9326 / 0.7817**	**0.9332 / 0.7839**
Attention NCF	0.9315 / 0.7786	0.9315 / 0.7783
Graph NCF	0.9304 / 0.7753	0.9310 / 0.7778
Basic NCF one-hot	0.9238 / 0.7558	–
Graph NCF one-hot	0.9232 / 0.7552	–

We present the best run for each type of model and input in Figure 4.11 and table 4.7. The first thing one might notice is how the train loss appears to suddenly increase every so often. This is because we follow the scheduling strategy we discussed in Subsection 3.3.2 for the hyperparameter w of Equation 3.9. That is, we start with a w of 0 (uniform negative sampling) and we increase it by 0.5 every 4-6 epochs (as indicated by the spikes in Figure 4.11) in order to sample harder and harder negatives. This should make the task more and more difficult as we go, resulting in the higher training loss we observe. Recall that the purpose of this is to sample more informative negatives instead of easy ones that are more likely to already be correctly ranked, resulting in vanishing gradients.

Unsurprisingly, the models using item and user profiles perform better than their one-hot counterparts. On the other hand, it is rather surprising that the Attention NCF model has been dethroned as the best model under this training setting. In the Figure 4.12, we can see how each model's validation NDCG when trained for the ranking problem (the dotted lines) versus when it was trained for the prediction problem (the solid lines). For most models, those two curves are relatively the same, except for the Attention NCF model where there is a significant improvement when trained for the prediction problem using the MSE loss.

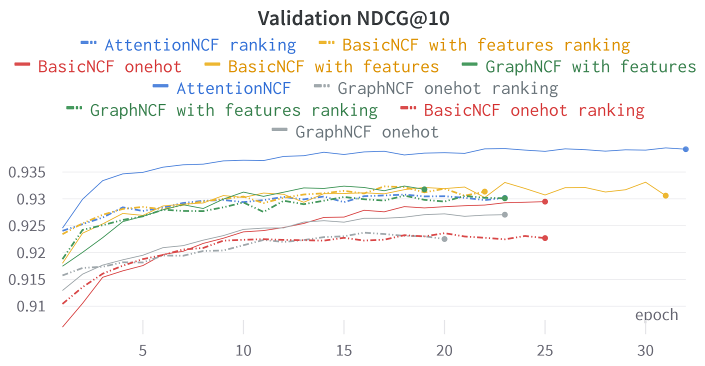


Figure 4.12. *Validation NDCG curves for the models trained for the prediction vs them trained for the ranking problem.*

All in all, whilst training for the ranking problem directly is indeed viable and it does require weaker supervision[10], it does not appear to offer any notable advantage over training for the prediction problem in this case. Conversely, it introduces more difficulties due to the need for negative sampling (which adds more hyperparameters to tune) and it is also more expensive computationally, since it requires two forward passes instead of one for each triplet. Rough estimates in minutes per epoch for the ranking models compared to the ones trained for the prediction problem are shown in Table 4.8.

Table 4.8. *Average time per epoch for the ranking models.*

Model	batch size	Average time with MSE (minutes / epoch)	Average time with BPR (minutes / epoch)
Basic NCF	512	2.037	3.227
Attention NCF	512	5.281	7.913
Graph NCF	512	5.714	10.826

4.3.7 Comparison with other methods as baselines

To get an idea of how our neural network models fair compared to other recommendation methods, we train and evaluate some simpler recommendation methods as baselines on our custom dataset by using, of course, the same train, validation and test sets. These simpler methods include:

[10] In many cases we might have access to ground truth for how to rank certain items but not to e.g. ratings.

- *Content-based similarity*: A purely content-based method, solving the ranking problem, where we use the cosine similarity between our user and item profiles to make top-k recommendations.
- *MF*: Matrix Factorization, a purely Collaborative Filtering method, as implemented by a third-party library[11] and tuned on the validation set by grid searching its hyperparameters (number of epochs, learning rate, factors dimension, regularization).
- *NCF-like MF*: Matrix Factorization, implemented by us akin to Basic NCF but without the MLP and by using the dot product of user and item embeddings directly for the final prediction. It can be used with both one-hot vectors and content-based profiles.

Table 4.9. *Test set evaluation for simpler baseline models and our own.*

Model	Test MSE	Test+ MSE	Test NDCG@10 / adjusted	Test+ NDCG@10 / adjusted
Content-based similarity	–	–	0.9165 / 0.7299	0.9167 / 0.7309
MF	0.5985	–	0.9247 / 0.7565	–
NCF-like MF one-hot	0.5761	–	0.9306 / 0.7744	–
NCF-like MF with features	0.5714	0.5665	0.9337 / 0.7846	0.9342 / 0.7862
Basic NCF one-hot	0.5749	–	0.9287 / 0.7712	–
Basic NCF with features	0.5657	0.5618	0.9320 / 0.7786	0.9327 / 0.7817
Attention NCF with features	**0.5244**	**0.5185**	**0.9387** / **0.8009**	**0.9396** / **0.8039**
Graph NCF one-hot	0.5858	–	0.9277 / 0.7690	–
Graph NCF with features	0.5667	0.5632	0.9321 / 0.7804	0.9324 / 0.7817

The results of these methods, compared to our own, are summarized in Table 4.9. There, we can see that:

- The pure content-based method performs significantly worse, as measured by NDCG, than both our one-hot and hybrid NCF models. It's possible that better similarity measures and/or user profile aggregations may improve these metrics, but we believe that there are potentially more patterns to be learned via Collaborative Filtering, assuming that we have access to such data, than by simply being based on user-item profile similarity.
- Our implementation of Matrix Factorization, as a special case of Basic NCF, performs better than the third-party library's implementation, possibly due to our use of early stopping and the use of Adam instead of SGD for gradient descend.
- More importantly, note how close the NCF-like Matrix Factorization model, a *linear* model, and Basic NCF are in performance. This is strong evidence to support that our dataset lacks significant non-linear and complex patterns for our deep learning models to capture. Hence, it's reasonable that even more complicated CF models like Graph NCF fail to achieve better performance on it. Attention NCF, on the other hand, must be able to achieve its better performance thanks to its item-item attention mechanism combined with its masking of the candidate item from the rated items.

[11] https://pypi.org/project/matrix-factorization/

Chapter 5

Conclusions

5.1 Conclusions

In this thesis, we have combined the Neural Collaborative Filtering framework [11] with content-based methods for item and user profiles, in order to acquire a hybrid recommendation system based on neural networks. This hybrid recommendation system is still mostly a Collaborative Filtering approach, as the training objective is that of Collaborative Filtering. Nevertheless, by incorporating content-based profiles for items and users we are able to avoid the cold-start problem, for which Collaborative Filtering methods are infamous, as well as leverage potential patterns in the content to achieve better performance.

We have proposed three increasingly complex deep learning architectures. Basic NCF is basically NCF but with fixed content-based user and item profiles instead of one-hot vectors. Attention NCF builds upon Basic NCF by creating the user profiles dynamically, during the forward pass of the neural network, using an item-item attention mechanism inspired from the work in [28]. Through this mechanism, it can attend differently to each interacted item from a user's known interactions depending on the candidate item it is trying to make a prediction for. Moreover, it allows us to present highly attended interacted items as an explanation for a recommendation of a certain candidate item, providing some much needed explainability. Finally, Graph NCF attempts to leverage the relationships of multi-hop neighbors in the user-item bipartite graph by incorporating Graph Neural Networks in the user and item embedding process, following the work of [29] and [10], starting however from fixed item and user profiles as in Basic NCF.

To train and evaluate these three proposed architectures we created a custom dataset of user-movie interactions by combining the popular Collaborative Filtering dataset of MovieLens with metadata (e.g. genres, actors, directors, etc.) for movies from the IMDb dataset. Using this dataset, we evaluated our models on both regression and ranking metrics. From our experiments, we concluded the following:

- Attention NCF outperforms both Basic NCF and Graph NCF. More specifically, we observed a $\sim$7.5% improvement in test MSE over Basic NCF. We believe that the reasons behind this improvement are the item-item attention mechanism we used and the ability to mask out the target training interaction from being used as input in the user profile construction during training (as described in Subsection 3.2.2).

- Attention NCF indeed offers a good degree of explainability through the interpretation of the item-item attention weights that are part of the constructed user profile, when making a prediction for a certain candidate item. This is intuitively shown in Subsection 4.3.5.

- Graph NCF does not seem to be any better than Basic NCF, despite being more sophisticated and arguably much more expensive as shown in Table 4.5 for both training and inference. We believe the reason to be that the collaborative signal in the user-item graph of *our* dataset simply isn't strong enough in order for explicitly capturing it via GNNs to be more beneficial than implicitly doing so through the objective function. In other words, there may not be complicated enough patterns in our dataset and our fixed user profiles to justify this complicated a CF model, regardless of what GNN architecture we use or any further hyperparameter tuning. This intuition seems to be supported by our experiments in Subsection 4.3.7, where a Matrix Factorization model (implemented as a special case of Basic NCF), a purely *linear* model, managed to achieve very comperable performance to that of Basic NCF. Attention NCF, on the other hand, works with dynamic user profiles via its item-item attention mechanism, where it is also able to mask the candidate item, achieving better performance in that way.

- Training our models for the ranking problem directly using the BPR loss, instead of the prediction problem using the MSE loss (or the BCE loss), is more costly, has more hyperparameters to configure (e.g. for negative sampling) and it also seemed to deprive Attention NCF of its superior performance as shown in Subsection 4.3.6. Therefore, we concluded that, given the choice, solving the prediction problem using a pointwise loss should be preferable.

Finally, we took our best Attention NCF model and we deployed it in a demo web application, in order to showcase and intuitively assess it. In this application, a user can input ratings to the movies we have available and then ask for recommendations. At that point, we collect the user's ratings and run inference on all the movies that he has not yet rated. After sorting these movies according to their predicted rating, we recommend to the user the top-K ones. For each recommendation, we also provide a list of the rated movies that have an attention weight that is sufficiently larger than we would expect a random rated movie to have (e.g. notably larger than $1 / |N(u)|$ from Equation 3.1) along with the attention weight itself. Using this list – when available – we can better understand why certain recommendations are being suggested.

5.2 Future work

Seeing as complicated models such as Graph NCF have failed to achieve notably better performance than Basic NCF – or even Matrix Factorization – in our dataset/task, it would be interesting to test our models on different CF datasets, assuming we have access to relevant item features for the items in them as well. Our hope is that there are complicated enough patterns out there that our increasingly complex deep learning models would be able to capture when other simpler models, like Matrix Factorization, would not.

When it comes to our dataset, we believe that our best bet for further performance improvement lies not in even more sophisticated and complex CF models, but in using more relevant item features. In our experiments, we have been relying on multi-hot encoded metadata for movies (e.g. genres, actors, etc.) from IMDb and the genome tag scores from MovieLens, as they were simple enough to extract for a large quantity of movies. However, the multi-modal content of the movies, as in their video, audio and text, has not been taken advantage of, when neural networks are universally praised for their ability to extract patterns from raw data such as images and text. Therefore, one interesting future work direction would be to attempt to incorporate these sources of information in our models.

To achieve that we may be inclined to add more neural network components in our models, e.g. a CNN to embed one or more images for each movie (e.g. a spectrogram of its audio or frames of the movie) or an LSTM or a transformer-based encoder to embed text (e.g. the movie's script). Doing so would allow us to train our models on these new features in an end-to-end fashion, using the same optimization objective we have been using so far. However, our models are already quite expensive by just using feature vectors and, given that the multi-modal content of a whole movie will be large, this approach will almost certainly be too computationally expensive to be viable, not to mention that it might face *vanishing gradient* issues should the network become too deep (e.g. with LSTMs). Instead, we will probably have to embed this content to dense feature vectors separately, using a different optimization objective, under an unsupervised setting (e.g. autoencoders). Having done that, we can simply add these dense vectors as part of the fixed item profiles that we have been using.

Other than that, further possible future work may include taking into account the rating of the rated item in the item-item attention mechanism of Attention NCF. So far, AttentionNet only uses the item embeddings of the candidate and rated items as input. But the rating of the rated item might also be an important factor to decide how much attention to give to the rated item. Therefore, we could try also including it or any other information we deem relevant in the input of AttentionNet (e.g. via concatenation).

Furthermore, as we noted in Subsection 4.3.1, message dropout did not really help in Attention NCF. However, that may be because we implemented a simple version of it, where the dropped edges are sampled uniformly from all user-item interactions in the batch, regardless of the user. As future work, we could try performing message dropout in a way that it is normalized for each user in the batch, e.g. randomly drop $p\%$ of the interactions of each user separately, to see if that version works better.

Last but not least, in our work we have considered the user's preferences static, in that it doesn't matter *when* a user interacted with what item. However, given our work on user profiles, there is a simple yet effective way to consider that. By heuristically calculating a weight that decays with time, we could assign a larger weight to items that were rated more recently than others. We can then somehow incorporate this weight in Equations 3.2 and 3.4. This should result in user profiles that better estimate the user's latest preferences, giving less importance to older ones, as the user's interests may change over time. Although, if we were to do this, we may want to also consider a more suitable train-val-test splitting strategy, where the latest interactions are left for the validation and test sets as explained in Subsection 4.1.3.

Bibliography

[1] Aggarwal, C. C. *Recommender Systems - The Textbook*. Springer, 2016.

[2] Bekker, J., and Davis, J. Learning from positive and unlabeled data: a survey. *Machine Learning 109*, 4 (Apr 2020), 719–760.

[3] Bell, R. M., and Koren, Y. Lessons from the netflix prize challenge. *SIGKDD Explor. Newsl. 9*, 2 (dec 2007), 75–79.

[4] Berg, R. v. d., Kipf, T. N., and Welling, M. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263* (2017).

[5] Burke, R. *Hybrid Web Recommender Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 377–408.

[6] Cai, C., and Wang, Y. A note on over-smoothing for graph neural networks. *arXiv preprint arXiv:2006.13318* (2020).

[7] Covington, P., Adams, J., and Sargin, E. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems* (2016), pp. 191–198.

[8] Funk, S. Netflix update: Try this at home, 2006.

[9] Hamilton, W. L. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning 14*, 3, 1–159.

[10] He, X., Deng, K., Wang, X., Li, Y., Zhang, Y., and Wang, M. Lightgcn: Simplifying and powering graph convolution network for recommendation, 2020.

[11] He, X., Liao, L., Zhang, H., Nie, L., Hu, X., and Chua, T.-S. Neural collaborative filtering, 2017.

[12] Kipf, T. N., and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[13] Koren, Y., Bell, R., and Volinsky, C. Matrix factorization techniques for recommender systems. *Computer 42*, 8 (2009), 30–37.

[14] Leskovec, J., Rajaraman, A., and Ullman, J. D. *Mining of Massive Datasets*, second ed. Cambridge University Press, 2014.

[15] Li, H. A short introduction to learning to rank. *IEICE Trans. Inf. Syst. 94-D* (2011), 1854–1862.

[16] Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[17] Lops, P., de Gemmis, M., and Semeraro, G. *Content-based Recommender Systems: State of the Art and Trends.* Springer US, Boston, MA, 2011, pp. 73–105.

[18] Ma, J., Li, G., Zhong, M., Zhao, X., Zhu, L., and Li, X. Lga: latent genre aware micro-video recommendation on social media. *Multimedia Tools and Applications 77*, 3 (2018), 2991–3008.

[19] Meng, Z., McCreadie, R., Macdonald, C., and Ounis, I. Exploring data splitting strategies for the evaluation of recommendation models, 2020.

[20] Oono, K., and Suzuki, T. Graph neural networks exponentially lose expressive power for node classification, 2021.

[21] Pan, R., Zhou, Y., Cao, B., Liu, N. N., Lukose, R., Scholz, M., and Yang, Q. One-class collaborative filtering. In *In ICDM 2008* (2008).

[22] Rendle, S., and Freudenthaler, C. Improving pairwise learning for item recommendation from implicit feedback. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining* (New York, NY, USA, 2014), WSDM '14, Association for Computing Machinery, p. 273–282.

[23] Rendle, S., Freudenthaler, C., Gantner, Z., and Schmidt-Thieme, L. Bpr: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618* (2012).

[24] Sarwar, B., Karypis, G., Konstan, J., and Riedl, J. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web* (New York, NY, USA, 2001), WWW '01, Association for Computing Machinery, p. 285–295.

[25] Seo, S., Huang, J., Yang, H., and Liu, Y. Representation learning of users and items for review rating prediction using attention-based convolutional neural network. In *International Workshop on Machine Learning Methods for Recommender Systems* (2017).

[26] Song, B., Yang, X., Cao, Y., and Xu, C. Neural collaborative ranking, 2018.

[27] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[28] Wang, H., Zhang, F., Xie, X., and Guo, M. Dkn: Deep knowledge-aware network for news recommendation, 2018.

[29] WANG, X., HE, X., WANG, M., FENG, F., AND CHUA, T.-S. Neural graph collaborative filtering. *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Jul 2019).

[30] WU, S., SUN, F., ZHANG, W., XIE, X., AND CUI, B. Graph neural networks in recommender systems: A survey, 2022.

[31] XUE, H.-J., DAI, X.-Y., ZHANG, J., HUANG, S., AND CHEN, J. Deep matrix factorization models for recommender systems. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (2017), IJCAI'17, AAAI Press, p. 3203–3209.

[32] ZHANG, S., TONG, H., XU, J., AND MACIEJEWSKI, R. Graph convolutional networks: a comprehensive review. *Computational Social Networks 6*, 1 (2019), 1–23.

[33] ZHANG, S., YAO, L., SUN, A., AND TAY, Y. Deep learning based recommender system. *ACM Computing Surveys 52*, 1 (Jan 2020), 1–38.

List of Abbreviations

ML	Machine Learning
NLP	Natural Language Processing
CF	Collaborative Filtering
NCF	Neural Collaborative Filtering
NGCF	Neural Graph Collaborative Filtering
NCR	Neural Collaborative Ranking
MF	Matrix Factorization
GMF	General Matrix Factorization
NeuMF	Neural Matrix Factorization
MLP	Multi-Layer Perceptron
NN	Neural Network
CNN	Convolutional Neural Network
LSTM	Long Short-Term Memory
GNN	Graph Neural Network
GCN	Graph Convolution Networks
GGNNs	Gated Graph Neural Networks
GAT	Graph Attention Networks
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
BCE	Binary Cross-Entropy
BPR	Bayesian Personalized Ranking
CG	Cumulative Gain
DCG	Discounted Cumulative Gain
NDCG	Normalized Discounted Cumulative Gain