ASSIGNMENT OF BACHELOR'S THESIS

Title:	Music Recommender System
Student:	Ondřej Šofr
Supervisor:	doc. Ing. Pavel Kordík, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2018/19

Instructions

Survey algorithms for recommedation of music. Focus mainly on collaborative filtering approaches and algorithms that can work with the temporal dimension (for example time events or sequence of genres). Design and implement a recommender system and evaluate the success rate in time context. Demonstrate functionality of the system on data provided by your supervisor.

References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 2, 2018

Bachelor's thesis

Music Recommender System

Ondřej Šofr

Department of Applied Mathematics
Supervisor: doc. Ing. Pavel Kordík, Ph.D.

June 28, 2018

Acknowledgements

I would like to thank my supervisor doc. Ing. Pavel Kordík, Ph.D. for giving me the opportunity to work on this interesting topic and for his guidance. I would also like to thank all members of my family for their endless support throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on June 28, 2018 ....

Czech Technical University in Prague
Faculty of Information Technology
© 2018 Ondřej Šofr. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Šofr, Ondřej. *Music Recommender System.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zabývá problematikou personalizovaného doporučování hudebních skladeb posluchačům. Jsou zde představeny přístupy využívané v současnosti, zejména metody kolaborativního filtrování. Důraz je kladen na zpracování časových informací o jednotlivých akcích uživatelů a jejich využití pro zkvalitnění doporučovacích systémů. Nejdůležitější částí je rozbor modelů predikujících aktivitu uživatelů. Je zde porovnána přesnost a výkonnost jednotlivých řešení i s ohledem na jejich využitelnost v praxi. Práce obsahuje výsledky experimentálního vyhodnocení představených metod nad daty reálných uživatelů.

Klíčová slova doporučovací systémy, kolaborativní filtrování, strojové učení, predikce časových řad, umělé neuronové sítě

Abstract

This thesis deals with the field of personalized recommendation of music. Modern approaches are described and analyzed, especially the methods of collaborative filtering. The main focus is the processing of temporal dimension data of user actions and its usage to improve recommendation systems. The most important part is the analysis of models predicting the activity of users. Prediction accuracy and efficiency of solutions are compared with emphasis on the usability in practice. This thesis contains experimental results of presented methods tested on real-world data.

Keywords recommender systems, collaborative filtering, machine learning, time series prediction, artificial neural networks

Contents

1 Introduction **1**
1.1 Goals of this thesis 2

2 Related work **3**
2.1 Recommender systems 3
2.1.1 Approaches 4
2.1.2 Evaluation 5
2.1.3 Music recommendation systems 7
2.2 Collaborative filtering 9
2.2.1 Representation of user-item interactions 9
2.2.2 User-based and item-based approaches 9
2.2.3 Processing of user feedback 11
2.2.4 Determining similarity of users or items 12
2.2.5 Model-based collaborative filtering 13
2.3 Machine learning 15
2.3.1 Key concepts 16
2.3.2 Overview of basic models 17
2.3.3 Ensemble models 18
2.3.4 Artificial neural networks 19
2.3.5 Recurrent neural networks 21

3 Time series prediction experiments **25**
3.1 Summary of used data 25
3.2 Data preprocessing 27
3.2.1 Discretization of time 27
3.2.2 Binning of activity 28
3.2.3 Transformation into a supervised ML task 28
3.3 Structure of experiments 32
3.3.1 Overall structure 32

3.3.2 Model behavior . . . 33
3.3.3 Measuring results . . . 33
3.3.4 Further data selection . . . 34
3.3.5 Implementation and environment . . . 34
3.4 Used models . . . 35
3.4.1 Baseline statistical models . . . 35
3.4.2 Perceptron model . . . 36
3.4.3 Gradient boosted trees model . . . 36
3.4.4 Multilayer perceptron model . . . 37
3.4.5 LSTM model . . . 37
3.5 Conclusion of results . . . 38

4 Advanced experiments **41**
4.1 Data preprocessing . . . 41
4.1.1 Selection of training data . . . 41
4.2 Used models . . . 42
4.2.1 Perceptron model . . . 42
4.2.2 Multilayer perceptron model . . . 42
4.2.3 LSTM model . . . 43
4.2.4 Stacked LSTM model . . . 44
4.2.5 Stacked GRU model . . . 44
4.2.6 Ensemble model . . . 44
4.3 Conclusion of results . . . 45

5 Proposed solution **49**
5.1 Description . . . 49
5.1.1 Structure . . . 50
5.2 Overview of usability . . . 50
5.2.1 Computational and memory cost . . . 50
5.2.2 Analysis of predictions . . . 51
5.3 Potential improvements . . . 51

Conclusion **53**

Bibliography **55**

A Acronyms **63**

B Visualizations of predictive behavior **65**

C Contents of attached storage medium **71**

List of Figures

2.1 Example of MLP structure 20
2.2 The inner structure of a LSTM unit (taken from [55]) 22

3.1 Distribution of playbacks of 14 random users during one day . . . 26
3.2 Binning preprocessing of raw data into an activity sequence 29
3.3 Initial extraction of samples 29
3.4 Example of persistence forecast prediction behavior 30
3.5 Advanced extraction of samples 32
3.6 Comparison of simple baseline models 35
3.7 Relative importance of features 37
3.8 Comparison of model performance with varying regularization settings 38

4.1 Comparison of model performance with varying number of LSTM blocks 43
4.2 Structure of ensemble model 46

B.1 Comparison of the behavior of LSTM model using different experimental settings 65
B.2 Example of predicting behavior of selected models 1 66
B.3 Example of predicting behavior of selected models 2 67
B.4 Example of predicting behavior of selected models 3 68
B.5 Example of predicting behavior of selected models 4 69
B.6 Example of predicting behavior of selected models 5 70

List of Tables

2.1 Example of user-item matrix 9
2.2 Confusion matrix 16

3.1 Comparison of the whole dataset and the selection of most active users 27
3.2 Comparison of performance of basic models 39
3.3 Comparison of relative count and importance of samples grouped by their sample weight determined during preprocessing 39

4.1 Comparison of performance of advanced models 47

Chapter 1

Introduction

We live in a period of time, where the popularity of digital content streaming services is constantly on the rise. This trend is even more evident at the field of music streaming. Major providers like Spotify, Pandora or Apple Music claim to have tens of millions of active users and those numbers are constantly increasing [1]. Most people prefer music streaming services over radio stations because they can conveniently choose what songs they want to listen to and this process is much more comfortable than having a music collection stored on CDs. Another advantage is the amount of content that can be easily accessed. All significant streaming service providers have collections containing tens of millions of songs. It is, however, very difficult for users to find relevant content in such an inexhaustible quantity of items. Because of that, providers try to offer convenient and personalized ways to discover music. This issue is mainly solved by recommender systems that suggest only such content that is likely to be considered useful by users.

A predominant technique used in modern recommender systems is *collaborative filtering*. The basic assumption of this approach is that there are groups of users with similar taste and listening behavior. Subsequently, there is high a probability that a certain user will like a content that is popular amongst a group to where he belongs. Recommender systems which use collaborative filtering usually provide a high accuracy of suggestions and they can be easily deployed no matter what type of content is offered. That is the reason why they are used in most systems that generate music recommendations. An important feature of collaborative filtering (in its basic form) is that it needs no other information than users' ratings of items. However, streaming services usually record and store many additional data about musical works and users. These pieces of information can be used to improve the quality of suggestions. Most companies try to utilize them but this is not a straightforward task and a lot of research is needed concerning this topic. That is the reason why this thesis focuses on utilizing time information of user behavior.

From a business point of view, it might be useful to predict when a particular user desires to listen to music. Such knowledge could be used to improve the user experience and to increase the service usage. For example, if a mobile application is able to display a notification with suggested song at the proper time, it could attract the users to listen to music more frequently. On the contrary, if a user is predicted not to be interested in such recommendations at a certain moment, the application would cease to display notifications to prevent dissatisfaction with the service. In terms of knowledge engineering, this problematics is closely related to the topics of machine learning and time series prediction. Utilizing a sequence of recorded data of past behavior, the goal is to predict user's activity in the future. Although time series prediction is a popular research field, most of the time experts focus on economical and insurance business topics such as stock prediction or risk management. This thesis is innovative because of its topic of music listening activity prediction and the usage of real-world data gathered by a music streaming service. Its main goal is to examine and compare the usability of prediction techniques used in other fields to accurately predict listening behavior of users.

1.1 Goals of this thesis

- Analyze machine learning approaches used for time series prediction and present possible solutions for user activity prediction task.
- Experimentally evaluate these approaches on real-world dataset. Determine model settings that bring the most accurate results.
- Analyze the usability of selected solutions in real-world recommender systems. If needed, propose changes to make such solutions more suitable.

The goal of this thesis, on the other hand, is not to create a standalone recommender system. Construction of such system is a well-researched task and a work of this size could hardly create any innovative outputs. Because of that this thesis focuses on a single specific topic where an improvement can be reasonably expected.

Chapter 2

Related work

2.1 Recommender systems

Recommender (or Recommendation) systems are information filtering software tools. Their main goal is to generate meaningful collections of suggested items for a particular user [2]. This behavior is extremely useful in all areas where the total number of possible retrieved options greatly exceeds the number of options that a typical user would consider to be interesting. Such examples are large online shopping websites like Amazon[1] or eBay[2] where millions of products are being sold but usual customer is likely to buy only a tiny portion of them.

The items can be recommended based on information such as their overall popularity or the demographics of the customer. However, modern systems usually perform an analysis of the past buying behavior of a customer to predict his future buying behavior [3]. Suggestions produced by recommender system are typically personalized for each user. The term *item* may stand not only for physical objects sold over the internet, but also for any digital content such as movies, videogames, music and even news articles or user-generated content on social networks.

Recommender systems are very important for companies that offer digital content directly to the customers. There are many goals that can be achieved by providing recommendation service (according to [4]):

- **Increase the number of items sold** The most important function of recommender system is to suggest items that consumer finds worth buying. Without these suggestions the user would have probably never discovered those items and his spending would be lower. This goal also applies if the provider does not profit directly from selling items but rather from some kind of periodic subscription fee or from advertising

[1] www.amazon.com
[2] www.ebay.com

revenue. In such cases it is important to keep the user inclined to use the service by showing him interesting content. The increase in profit can be really significant. Netflix [3], one of the biggest video-streaming providers, reports that 75 % of user views are made as a result of their recommendation features [5].

- **Sell more diverse items** Without personalized recommendation system the service provider has to be more conservative about offering less-popular products to consumers as those can be expected to be bought less frequently. But not selling such long tail products might be a missed opportunity and users usually like to discover novel items.

- **Increase the user satisfaction** A typical user expects to get interesting and relevant suggestions and user satisfaction is a vital part of provider's success. That is especially important for cases where the provider does not sell items or digital content, for example news websites. Good recommendation engine will show interesting articles to user, who will stay on the website much longer than he normally would. That may increase advertisement revenues for such website.

- **Understand the user better** Information about customers' desires is invaluable for every company. It may be beneficial for logistic and management planning and to estimate potential interest in future products. When a new product enters the market it might be possible to use recommendation system to find a subset of users who are most likely to find the product interesting – this is called *inverse recommendation* [6].

2.1.1 Approaches

Although the general goal of producing personalized suggestions is shared amongst all recommender systems, there are many ways how to achieve it. Systems can be divided by their basic approach into three groups (as categorized in [7]:

- **Collaborative filtering approach** This approach focuses on user-item interactions. Every user has his observed behavior consisting of purchases, ratings or views of items. If a group of users share their interest in a specific item, it is reasonable to suggest that item to other users with similar behavior. This approach tends to bring good results and it is capable of suggesting novel and serendipitous items. The system needs no additional knowledge about the specific domain as all needed information is observed from user behavior. A more detailed review of collaborative filtering can be found later in this chapter.

[3] www.netflix.com

- **Content-based filtering approach** Such systems create suggestions based on items that a user found interesting in the past. Using known attributes of these, they try to find similar items. Other information about user may also be taken into account. The result is a relevance judgment that represents the user's predicted level of interest in particular items. Unlike collaborative filtering based systems, there is no need for information about other users and their behavior. This makes content-based approach more suitable for tasks where there is not enough data about user behavior (the so called *cold start problem*) [8]. However, a good knowledge of items is needed for those recommender systems to operate effectively. Also, as most of them use textual features to represent items and user profiles, they might suffer from the classical problems of natural language ambiguity [9]. Another drawback is their lack of ability to suggest completely novel items.

- **Hybrid approach** Recommender system can be created by using multiple approaches. A good example would be a system that uses content-based filtering for a particular user when there are not enough users with similar behavior and switches to collaborative filtering once there are. Hybrid recommender systems can maintain advantages of other categories and limit their disadvantages, making them very efficient.

Some sources like [10] also consider other approaches to be important independent categories. Knowledge based systems take explicit user requirements and search the set of available items to find a best match. Demographic recommender systems usually combine user's demographic information with other context to make suggestions. Both of these approaches do not need large datasets of user-item interactions. It may be beneficial in some areas to include those approaches into hybrid systems or into ensembles of recommender systems.

2.1.2 Evaluation

Important topic concerning recommender systems is measuring their performance. Generally speaking, the ultimate goal of every recommender system is to increase *conversion rate*. This metric is defined as the ratio of users who take a certain action depending on provider's business goals (e.g. visit a specific page, listen to a song or make a purchase) to the total number of users who receive a specific cue (e.g. being suggested an item by a recommender system) [11]. High increase of conversion rate after recommender system deployment indicates that it is successful in creating valid suggestions because users are more attracted to do actions utilized by the system provider. There are also many other metrics that may be observed.

2.1.2.1 Offline evaluation

There are three basic ways of evaluating recommender system performance. The first is offline evaluation. Collected data recorded from the past are split into two parts – *training set* and *test set*. The recommender system can only use knowledge of the first one to make predictions about values in the second one. Such predictions are then compared with recorded values in the test set. This approach is clearly the easiest as it requires no additional testing done by real users. That means it is very cheap to test any changes done to the system because the only thing that is needed is computation power and time. Because there is no additional user interaction, offline evaluation is suitable for comparison of different recommender systems. This makes it popular in academic research as all results are easily reproducible and proposed outputs can be compared with other solutions. However, the main drawback is that the measured performance is often misleading. Even when using appropriate metrics and concluding the testing phase properly, a recommender system that performed well in offline evaluation may be drastically less successful when deployed to real-world usage. Most of the time it is due to the fact that user behavior captured in testing data is insufficient for proper modelling of future behavior of users [12]. Nowadays, offline evaluation is often considered to be a basic auxiliary approach and enterprise recommender systems are additionally tested in different manners before deploying.

2.1.2.2 User studies

Another option is to gather feedback from a group of participants of a controlled study. This is an approach popular in marketing business which can be also used in this field. Outputs of various recommender systems are presented to users who evaluate them. This type of studies is expensive and may provide biased results. It is difficult to correctly select a representative sample of users to participate in study and even the behavior of such users can be affected by the fact that a rating is expected from them. The advantage of user studies is that the provided feedback is much more detailed than any output from the other two approaches which can be crucial to understand the current solution's strengths and weaknesses (as pointed out in [13]).

2.1.2.3 Online evaluation

Online evaluation is perhaps the most accurate way of determining the quality of a recommender system. Such system is simply integrated into existing production infrastructure like website or mobile application and the changes of user behavior are observed. Performance is then measured as an increase/decrease in an appropriate metrics like conversion rate or total revenues from sold items. While this may be a straightforward approach, there are many issues connected to it. To avoid corruption of results by outer factors

like changes of user behavior during the year, such experiments are usually performed as *A/B tests* [14]. That means users are divided into two groups – the first uses a certain baseline recommender system (typically an already tested and used) and the second uses the one that is to be evaluated. It is crucial to choose these two groups right to avoid influencing the results, for example by running a series of *A/A tests* first (observing the behavior of both groups in identical conditions). It is also quite dangerous to perform an online evaluation from a business point of view because customers might be dissatisfied with the tested system's suggestions. Because of that, the group of testing users is usually tiny in comparison to the group of all users. The biggest advantage of online evaluation is the fact that users do not know their behavior is used as a part of testing at all, so the results are unbiased and valuable. In production it is usually used as a second phase of evaluating changes of recommender systems, performed after successful offline testing [15].

2.1.3 Music recommendation systems

Although listening to music seems to be an activity similar to watching movies or reading web articles, there are a lot of differences that has to be taken into account. A consumption time of one song is much shorter than consumption time of one movie. An observation related to this is that people can often decide whether they like the song or not after a few opening seconds, while they need a much longer time to be able to rate a movie. Sound is also perceived differently than pictures by human senses, which may result in specific behavior patterns. For example, music is often listened to as a *background noise*, which means that the user is not paying full attention to it. The most important difference between music and other types of content from a recommender systems point of view is that users are much more likely to consume the same item multiple times [16]. People are inclined to repeat their favorite songs even in the same listening session, which is a behavior that is unexpected when reading news articles, for example.

Another property of audio content is that the sound can be analyzed with a variety of methods. Music features like beat (tempo), dynamics, key or chord distribution can be extracted from audio tracks and used for content based recommendation [17]. With additional metadata collected about songs and artists, using content based recommender systems can be a viable choice, as shown in the next subsection on the example of Pandora music streaming service. Collaborative filtering techniques are inherently *domain-agnostic*, so they can be easily applied here as well, but there are several issues. Explicit ratings are relatively rare and recorded data tend to be sparser, which makes collaborative filtering a less dominant approach than in other domains (according to[18]).

The format of recommendations may also be a bit different. In many other areas, a set of items is selected by the recommender system and presented

to the user at the same time. On the next occasion a new set of items is generated and so on. However, many music recommender systems are built to predict a sequence of songs rather than just a set of them. The result is slightly different from predicting one item per step as the system has to create a balanced playlists where the order of songs ensures a pleasing experience. There are many approaches to playlist construction and the topic is frequently researched (for example in [18]).

2.1.3.1 Examples of music streaming services

The nature of music recommender systems can be observed on two streaming services with the largest number of active users – Pandora Radio and Spotify. Both companies have surprisingly different approach to this task.

Pandora Radio (or simply Pandora) is a USA based audio content streaming provider that resembles internet radio stations. Each user receives a personalized stream of songs selected by engine built around the *Music Genome Project* [19]. This is a complex labeling process with precisely defined metodology. A musician or a group of musicians carefully listen to a song and manually submit ratings of hundreds of musical features, called *genes*, such as the level of distortion on the electric guitar or the type of background vocals. A content-based recommender system is then used to suggest content with similar musical genes as the one that a user likes. The result is a playlist consisting of songs that can be rated as satisfactory or unsatisfactory, which changes the importance and preferred values of individual genes for future predictions.

Spotify is the world leading music service provider, surpassing Pandora in 2016 [20]. Spotify's feature *Discover Weekly* is highly praised by its users as one of the best ways to explore the musical world. This feature focuses on providing novel recommendations by combining three model groups (as described in [21]). The first one is a group of models using collaborative filtering which utilizes user behavior. The second one consists of natural language processing models that are used for sentiment analysis of articles, blogposts and discussions about specific artists and songs that are scraped from the entire internet. And finally, there are convolutional neural network models that analyze raw audio data. Outputs of these three categories of models are combined to provide accurate suggestions. This approach is very robust and can be used even in cases where individual models perform poorly due to unfavorable circumstances, for example a lack of data.

2.2 Collaborative filtering

Collaborative filtering refers to a class of techniques used in recommender systems, that recommend items to users that other users with similar tastes rated positively in the past. The basic assumption is that if two users share their opinion on an item, they are more likely to have similar opinion on other items than two randomly chosen users.

2.2.1 Representation of user-item interactions

To apply this approach, three things are needed – a set of users U, a set of available items I and the historical data for each user concerning his interactions with certain items. The most common way of data representation for collaborative filtering purposes is the *user-item matrix*. Traditionally, each row represents interactions of one user and each column represents interactions made by all users with one particular item. This can be specified as a matrix R, in which the value of $R_{i,j}$ denotes the preference of user $i \in U$ to item $j \in I$ (as in [16]). Values of this matrix are usually numerical representations of user ratings of items. More about their meaning and representation can be found in the section 2.2.3 of this thesis. An example of user-item matrix is shown at 2.1. Notice that this matrix is rather sparse, i.e. there are a large number of unspecified values. That is nothing unusual as a typical user interacts only with a small portion of items. Real-world dataset matrices can be even much sparser.

Table 2.1: Example of user-item matrix

	A	B	C	D	E
Alex	3	7	-	3	3
John	6	10	3	-	-
Patrick	10	-	1	2	-
Susan	-	-	-	8	9
Mary	5	-	3	1	-
Helen	7	-	6	-	8

2.2.2 User-based and item-based approaches

As stated earlier, collaborative filtering utilizes interaction data collected from users. In order to make a suggestion, it is necessary to determine how likely a particular user is to be satisfied with a particular (previously unseen) item. Generally, this can be done by examining the user-item matrix in two basic manners:

- **User based approach** When determining $R_{i,j}$ where $i \in U$ and $j \in I$, the first step is finding users with similar behavior who rated the item

j. The value $R_{i,j}$ is then computed from values $R_{u,j}$ where $u \in U_S$ and $U_S \subset U$ is a set of users with similar behavior as user i.

A following simplified example explains this approach. Let's try to determine, using user-item matrix in Table 2.1, how is user Mary expected to rate item B. The goal is to find a group of users that rated item B and their behavior is similar to Mary's behavior. User John seems to belong to this group, as he rated items A and C in a closely similar way as Mary and there are no other items rated by both. User Alex also rated item B; however, his other ratings are not significantly similar to corresponding ratings made by Mary. Because of this, only John should be considered to have similar behavior as Mary. As John rated item B with a value of 10, Mary can be expected to like this item and rate it with a high value as well.

- **Item based approach** In comparison to the former approach, the first step is to find a group of items similar to the item of which the rating is being predicted. Similar items are those that are generally rated in the same way by all users. Item attributes are not compared, only relations with users – otherwise that would lead to a content based (or hybrid) approach. The value $R_{i,j}$ is computed from values $R_{i,k}$ where $k \in I_S$ is an item from a group of similar items $I_S \subset I$.

  Let's try to determine, using Table 2.1 again, how is user Helen expected to rate item D. Helen rated items A, C and E. First step is to choose a subset of items, which other users rated similarly as item D. Item A does not really suit this as Patrick rated it with a value of 10 and yet he rated item D with a value of 2. Item C is more suitable because both Patrick and Mary rated it with rather low values and so did they rated the item D. Item E can also be considered to be akin to item D since both Susan and Alex rated them each with similar values. (Notice that while Alex dislikes both items, Susan likes them. This causes no issues at all when using collaborative filtering.) It was decided to consider items C and E to be similar to D. Helen rated them with the values of 6 and 8 respectively. One possibility is, for example, to compute their mean – which is 7. This value can be considered to be Helen's expected rating of item D.

These examples are just illustrative and similarity of two users or items is determined intuitively. The exact ways how to measure similarity are described in section 2.2.4. In order to decide which users should be taken into account, k-NN algorithm (presented in the section 2.3.2) is often used.

2.2.3 Processing of user feedback

The user-item matrix is a key element of collaborative filtering. To construct a successful recommender system, it is crucial to ensure that the user-item matrix is filled with meaningful and useful values. This data can be obtained in various manners, which are divided into two main categories (as in [22]).

- **Explicit rating** is gathered by prompting users to consciously rate certain items. An example is a 5-star rating system available at Internet Movie Database [4]. Users can rate movies by selecting 1 to 5 stars (5 being the best rating) while adding an optional text. Both of these pieces of information are considered to be explicit rating because they are provided intentionally by the user. Another example is giving an item an *I like it* label, as known from most social networks.

- **Implicit rating** is gathered by learning from users behavior over time [7]. For example, in a music recommender system, if a user listens to a track several times the system may infer that the user has an interest in that track (as used in [23]). Other examples might be page visits or purchases – generally all user actions by which the user is not intentionally rating the item. Purchasing an item is intuitively a strong indication of user's interest, but it might be possible that the user is buying something as a gift and he actually does not like the item. Because of this it cannot be considered to be an explicit rating.

Although explicit rating is generally seen as more valuable [22, 24], it is also much more difficult to obtain. Users are reluctant to do tasks (like rating) that require even a minimal effort. Moreover, these ratings might be biased, as users usually rate items only on specific occasions. Many customers rate services only when they are unsatisfied with it [12]. A difficult task is to populate the user-item matrix with values using implicit feedback. It usually has a complex structure consisting of various observations. The most basic approach is to compute a rating from this data, which the user would most likely assign to an item, should he rated it (proposed by [25]). An example would be a music recommender system that computes the ratio of completed playbacks (e.g. when the user chose to listen to this song until the end) to all playbacks (including those that the user chose to stop before the end) for each user and item. The resulting values would be real numbers from the interval 0 to 1, with higher values signifying better estimated rating. If no playback of an item was made by certain user, the corresponding value can be treated as unknown (the so called *All Missing as Unknown* approach, shown in [26]). There are however many more sophisticated approaches and many researchers focus on this topic (a complex overview can be found in [27]). There are a large number of studies concerning implicit feedback in the

[4] www.imdb.com

field of music recommendation systems. For example [28] focuses on finding correlated explicit and implicit rating actions and [29] exploits the usage of time-related context of implicit feedback.

2.2.4 Determining similarity of users or items

As shown in the previous examples, the key part of collaborative filtering algorithm is determining the similarity of two users (or items in item-based approach). It can be easily deducted from the shape of user-item matrix that this task is equivalent to computing the similarity of vectors of the same length. Two vectors containing information about the user interactions with items are taken from the user-item matrix and a resulting value is computed using a similarity function.

There are many similarity functions and a proper one has to be chosen for each system with respect to the domain and structure of data in user-item matrix. The commonly used ones are (according to [30]):

- **Cosine similarity** measures cosine of the angle between two vectors. The resulting value is in the range $[-1, 1]$, or $[0, 1]$ in case only non-negative values are present in the user-item matrix. A higher value means that the two vectors are more similar to each other. The exact computation for vectors $\vec{a}$ and $\vec{b}$ of dimension n is shown in equation 2.1. The symbol "$\cdot$" stands for the Euclidean dot product of two vectors. The entire vectors can be used in computation, provided that the missing values are replaced by zeroes.

$$\text{Cosine similarity}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}} \tag{2.1}$$

- **Pearson correlation coefficient** (commonly represented as r) measures the extent to which the corresponding values in two vectors are correlated. The resulting value is in the range $[-1, 1]$ and a high value indicates close similarity. When using this method, dealing with missing values could spoil the results [30]. Because of that, only *co-rated* parts of vectors (parts where both vectors contain known values) are used. The exact computation for vectors $\vec{a}$ and $\vec{b}$ and M, which is a set of indices where both $\vec{a}$ and $\vec{b}$ contain known values, is shown in 2.2.

$$r(\vec{a},\vec{b}) = \frac{\sum_{m\in M}(a_m - \bar{a})(b_m - \bar{b})}{\sqrt{\sum_{m\in M}(a_m - \bar{a})^2}\sqrt{\sum_{m\in M}(b_m - \bar{b})^2}}$$

$$\bar{a} = \frac{1}{|M|}\sum_{m\in M} a_m \qquad \bar{b} = \frac{1}{|M|}\sum_{m\in M} b_m \tag{2.2}$$

Pearson correlation coefficient can be used to compute both the similarity of two items or two users. However, cosine similarity is not suitable for computing item similarities (according to [31]) because different users might use different rating scales. This is addressed by using the *adjusted cosine similarity*, which subtracts the corresponding user average rating from each co-rated pair (further described in [30]). Adjusted cosine similarity in fact has almost identical formula as the Pearson correlation coefficient [31]. This shows that the previously explained similarity metrics are related to each other.

2.2.5 Model-based collaborative filtering

So far in this chapter, only *memory-based* recommender systems were described. They are characterized by having the entire user-item matrix stored and using its values for calculations of predicted ratings. This approach has its limitations in real-world usage. The crucial problem is scalability. A memory complexity of user-item matrix in its basic form is $\mathcal{O}(nm)$ where n is the number of users and m is the number of items in the dataset. This means that it is not suitable for systems with large number of users and items. Although this might be mitigated to some extent by using storage structures for sparse matrices, another problem is high computational cost of operations over such matrix.

Because of this, it is often more suitable to construct an approximate model of such system that stores less data and computes recommendations more efficiently, even with the risk of being less accurate. These so called *model-based* recommender systems often utilize matrix decomposition methods known from the field of linear algebra. Generally, the user-item matrix is factorized into several smaller matrices. The product of such matrices is then the approximation of user-item matrix.

A notable method is the *singular value decomposition* (SVD). Widely used in the field of information retrieval, SVD uses three matrices to map both users and items to a joint *latent factor* space. Latent factors (also called *features* or *concepts*) are traits generated from the data that describes some shared characteristics of items (though they are mostly uninterpretable).

SVD factorizes the user-item matrix R in the following way (taken from [32]):

$$R = U \Sigma V^T$$
$$R \in \mathbb{R}^{n \times m} \quad U \in \mathbb{R}^{n \times n} \quad \Sigma \in \mathbb{R}^{n \times m} \quad V^T \in \mathbb{R}^{m \times m} \tag{2.3}$$

where the matrix U consists of the orthonormalized eigenvectors of RR^T and the matrix V consists of the orthonormalized eigenvectors of $R^T R$. The matrix Σ is a rectangular diagonal matrix with nonnegative real numbers. The diagonal elements of Σ are the non-negative square roots of the eigenvalues of $R^T R$ (and RR^T as well) called *singular values*. These values are sorted in decreasing order on the diagonal, i.e. $\Sigma_{1,1} \geq \Sigma_{2,2}$ and so on.

The i-th latent factor is described by the i-th columns of matrices U and V and singular value $\Sigma_{i,i}$. The corresponding singular value signifies the importance of such latent factor. To lower the size of used matrices, parameter $c \in \mathbb{N}, c \leq min(n, m)$ is introduced. Only the c most important latent factors are preserved and the matrices U,Σ and V are altered so they contain only the data corresponding to these latent factors. The matrix $\hat{R}$ is an approximation of user-item matrix R with the same size, defined as:

$$\hat{R} = U \Sigma V^T$$
$$\hat{R} \in \mathbb{R}^{n \times m} \quad U \in \mathbb{R}^{n \times c} \quad \Sigma \in \mathbb{R}^{c \times c} \quad V^T \in \mathbb{R}^{c \times m} \tag{2.4}$$

Only the resulting small matrices U,Σ and V are stored. These three matrices describe a space of dimensionality c. Further computations like searching for similar users then take place in this reduced space instead of the original one, which greatly improves their efficiency. It was experimentally shown on many occasions (for example [33]) that even a small value (i.e. <100)of c is sufficient to maintain accuracy of the approximation and thus the computational improvement over memory-based approaches is substantial. Reduction of dimensionality can sometimes make the model even more accurate by increasing its robustness. However, there are several problems with SVD in the collaborative filtering domain. SVD has limited ability to process missing values and the computational cost can be also an issue (both discussed in [34] and [33]).

Another popular matrix decomposition algorithm in the field of recommender systems is the *UV decomposition*. An approximation of user-item matrix R is constructed in the following manner:

$$R \approx \hat{R} = UV^T$$
$$R, \hat{R} \in \mathbb{R}^{n \times m} \quad U \in \mathbb{R}^{n \times c} \quad V^T \in \mathbb{R}^{c \times m} \tag{2.5}$$

where $c \in \mathbb{N}$ is a parameter that determines the reduction level, which usually is a rather low value (it has a similar behavior as the parameter c in

SVD). To get a predicted rating which a user is expected to give to an item, the dot product of the two corresponding vectors is used, i.e.

$$\hat{R}_{ij} = \sum_{k=1}^{c} (u_{ik} v_{kj}) \tag{2.6}$$

Values in U and V do not have the strict mathematical meaning as the values in matrices constructed by SVD. There are various ways how to obtain U and V, but the most common (according to [35]) is to initialize them randomly and then iteratively adjust the values to minimize the difference between matrices R and $\hat{R}$ (which can be measured for example as the sum of absolute errors, i.e. differences between each pair of corresponding values in the matrices). Such numerical approximation method, aiming to find a local minimum of difference, is called *gradient descent*. Another method (described in [36]) is the *Alternating Least Squares*, which is based on temporal fixation of certain values and computing the rest by the *least-square* technique.

2.3 Machine learning

Machine learning is an extensively researched topic in the field of *artificial intelligence* (AI). It focuses on solving problems by giving computers the ability to learn from data without being explicitly programmed for a specific task. This is the key strength of this concept because many real-world tasks are too demanding to be solved by humans efficiently or at all. The goal of machine learning is to construct algorithms which are able to find patters in the provided data, gather the knowledge and utilize this process when facing new challenges in the future.

Typically, a model is trained by inputting multiple data *samples*, which are instances of a problem that the model is expected to solve. Machine learning tasks can be divided into three groups depending on the format of the training process (summarized from [37]):

- **Supervised learning** The model is given samples containing *labels*. Labels are desired outputs attached to each sample and therefore the model can deduce what is its expected behavior An example of such sample is an image labelled with a description. A suitable model can be expected to learn how to describe unknown images if it is provided with enough training samples.

- **Unsupervised learning** During the training process only task instances without labels are provided to the model. This is a useful approach in situations where there are no desired outputs that are known beforehand. The model is left on its own to find patterns in the data. This method can be used for tasks like *clustering*, i.e. finding groups of similar instances.

- **Reinforced learning** This is a hybrid approach combining both previous methods. The model is not given a fixed set of samples as during supervised learning, but instead it can perform actions and observe how such actions are rated. The model is thus motivated to explore various possible solutions. Reinforced learning is often able to solve certain difficult tasks much better than any other method as the models are able to come up with highly innovative behavior. It is the key aspect of the highly successful AI program AlphaGo Zero [38].

Because only supervised machine learning is used in this thesis, the rest of this chapter describes aspects connected to this approach.

2.3.1 Key concepts

There are several concepts related to supervised machine learning that need at least a brief explanation before moving on to more advanced topics. They are described in this section.

Tasks solved by machine learning can be split into two main groups – *classification* and *regression* tasks. The difference is in the format of output variables. When solving classification problems, each instance belongs to one class. The output is thus a discrete (categorical) variable. An example is a model that labels each song with a genre tag. If there are only two possible classes, it is a special case called *binary classification problem.* Regression problems on the other hand allow the output to be any numerical variable. An example of such would be a model predicting the salary of users by their shopping behavior.

While this difference seems to be marginal, it determines the way of evaluating such models. For binary classification problems (for example predicting a condition whether or not a user will like certain songs) a *confusion matrix* is constructed as shown in table 2.2.

Table 2.2: Confusion matrix

Condition:	Observed	Not observed
Predicted	TP	FP
Not predicted	FN	TN

TP stands for the number of true positive samples (condition was predicted and observed). Correspondingly, FP stands for false positives, FN for false negatives and TN for true negatives. Several widely used metrics can be computed from this table, for example *precision*:

$$Precision = \frac{TP}{TP + FP} \tag{2.7}$$

Some binary classification systems do not provide strict categorical label, but rather a probability that an instance belongs to a certain class. Such

scoring can be used with a *threshold* value to produce a discrete binary classification [39]. Altering the threshold changes the distribution of instances in the confusion matrix. Because metrics like precision change with the varying threshold, it is not convenient to use them to evaluate the performance.

A suitable approach is to use a *receiver operating characteristics* (ROC) graph. This is a two-dimensional graph where the *true positive rate* (TPR, called also *recall*) is plotted on the Y axis and *false positive rate* (FPR) on the X axis (TPR and FPR computation is described in figure 2.8). Every value of threshold can be depicted by a corresponding point in this space. If connected, such points for various threshold settings form a ROC curve which is a common visualization method.

$$TPR = \frac{TP}{TP + FN} \quad FPR = \frac{FP}{FP + TN} \tag{2.8}$$

To get a numerical evaluation of performance, *area under ROC curve* (AUC) is used [39]. As both true positive rate and false positive rate have range of $[0, 1]$, AUC has also the range of $[0, 1]$. Higher value means better model, while a completely uninformed one (which predicts randomly) is expected to have an AUC of 0.5.

Regression tasks are usually evaluated by an error function that takes into account how different are the predictions from observed data. Examples are the *mean absolute error* (MAE) or *root-mean-square error* (RMSE), which are discussed and compared in [40].

The construction of machine learning models can be divided into two phases – training and testing. During training phase the model constantly improves its performance by learning from the provided data. During testing phase the model is provided with previously unknown instances of the problem and its performance is measured. This brings the question of how the model should be trained to perform the best not only on training data, but also on test data. Several complex models can be trained to have a perfect performance during training, but this comes at the cost of poor generalization ability and bad performance on previously unseen data. This is known as *overfitting* and is generally signalized by much larger errors on test data than on training data. Because testing performance is sometimes difficult to obtain (it might be gathered by expensive online tests), a subset of training data called *validation data* is often not presented to the model during training. Validation data can be used to simulate a previously unseen data, by which the model's behavior can be observed. This helps to discover better model settings before advancing to the testing phase.

2.3.2 Overview of basic models

A short description of the most common models (or model families) is presented in this section. As the machine learning tasks can be very diverse, each

model has its usage in some fields depending on its advantages and disadvantages.

- **Linear regressor/classifier** This is probably the simplest model in machine learning. Input features of instances are given weights depending on an evaluation (loss) function, thus the output is a linear combination of inputs. These weights can be determined by statistical methods like *least square* technique or by *gradient descent* method.
- **Nearest neighbors** The k-nearest neighbors algorithm (k-NN) is based on the assumption that similar instances have similar labels. Using a selected similarity function, it finds a set of k (which is an adjustable parameter) most similar instances and infers the result from their labels.
- **Naive Bayes classifier** This model utilizes the statistical background of Bayes' theorem. It assumes that all input features are independent of other features (thus called *naive*). This approach brings good experimental results [41] and it is also a rather simple model, which needs very few training samples to be functional.
- **Decision trees** This is a large family of models which utilize a tree-like structure. They can be interpreted as a sequence of *if-else* rules, which makes them understandable from a human point of view. Decision trees can be also easily utilized in *ensemble* models.
- **Artificial neural networks** (ANNs) are a large family of models, whose structure loosely models the neurons of a human brain. As they can extract complex patterns and solve difficult tasks, neural networks are very popular topic of modern machine learning research. ANNs are thoroughly described in the following sections.

2.3.3 Ensemble models

Ensemble modeling is a highly successful approach, which led to victory in the famous recommender systems competition – the Netflix prize [42]. Ensemble models combine several simpler models to utilize their advantages and bypass their limitations. Generally, the combination of models (either of the same type but with diverse behavior or completely different) lead to models resistant to overfitting, yet capable of finding complex patterns [43]. There are three basic techniques of ensembling:

- **Bagging** (bootstrap aggregation) uses a large number of simple models that are trained in parallel. To avoid an unwanted situation in which all models are too similar to each other, a different subset of training data (samples are chosen randomly) is provided for each model. To aggregate a final result, *voting* for classification tasks and *averaging* for regression

tasks are used. An example of bagging are *random forest* models, utilizing decision trees with additional random settings to support their diversity (described in [44]).

- **Boosting** also uses many simple models (called *weak learners*), but those are trained in sequence. The most important concept is that samples that are not successfully predicted by previous weak learners gain larger weights for future training [45]. This tells the next models that such samples should be prioritized. An example of this technique is *gradient tree boosting*.

- **Stacking** is a technique that combines multiple models via a *meta-classifier* or *meta-regressor*. The base models are trained as usual and then a meta-model (sometimes called *stacking model*) is trained using their outputs. Unlike in previous approaches, base learners are often complex and heterogeneous, i.e. constructed by different algorithms [46]. Meta-model can theoretically be any model, but ensembles of decision tree and neural networks often outperform simpler models as proved in many kaggle [5] competitions, for example [47].

2.3.4 Artificial neural networks

Although the concept of artificial neural networks dates back to 1950s [48], only with the recent improvements of computer performance they became a truly dominant machine learning model family. The main idea of ANNs is to simulate the structure of organic brain and its decision-making processes.

The basic unit of ANNs is an artificial neuron unit. It has several inputs from which the output is computed in the following manner:

$$y = \phi(\sum_{j=1}^{n} w_j x_j + b) \tag{2.9}$$

where n is the number of inputs, b is a bias, w_j is the weight attached to input j and x_j is its value. ϕ is an activation function (for example a sigmoid function), which is used to keep the output in some reasonable range suitable for consequent processing (which is especially important in complex NN structures).

2.3.4.1 Perceptron

The simplest NN model is the perceptron, which consists of a single neuron. Initially it was constructed only for classification and thus a *Heaviside step function* (yielding 0 for negative argument and 1 for positive one) was used. Nowadays it is often used with a continuous function, allowing perceptron to

[5] www.kaggle.com, a website hosting machine learning contests

be used in regression tasks. The computational ability of perceptron is very low, as it behaves like a linear classifier and cannot achieve zero error if the instances of data are not linearly separable.

2.3.4.2 Multilayer perceptron

Multilayer perceptron (often called *feedforward neural network*) is a complex model containing large amount of neurons. These neurons are structured into several layers, where each neuron is connected to all neurons in the next layer as shown in figure 2.1. The instances are inputted into the first (input) layer and the created values are propagated in one direction through all the layers of the network, using the standard neuron structure with weights, bias and activation functions. The key parts are the hidden layers which significantly improves the descriptive ability of this model. MLP is capable to find non-linear patterns and to solve difficult tasks surprisingly well. The popular term *deep learning* is connected to the fact that modern MPLs have many hidden layers and thus a high depth [49].

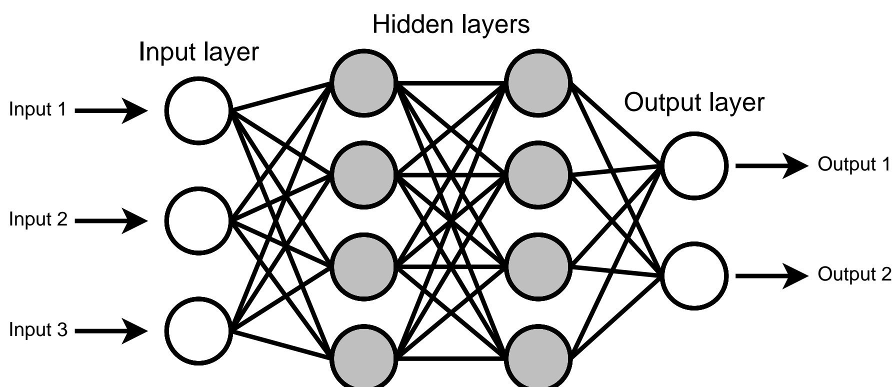

Figure 2.1: Example of MLP structure

2.3.4.3 Backpropagation

The behavior of neural networks is dependent on a large number of parameters – weights and biases of neurons. The process of learning therefore aims to find the best set of such parameters. Neural networks are trained using a gradient descent optimization method, meaning that the parameters are first chosen randomly and then iteratively adjusted in order to minimize the error (loss) function. An effective method of such optimization capable of training even very complex NNs is called *backpropagation*.

The key part of this algorithm is that after evaluating an instance, the error is computed and propagated backwards through the network. This way,

a contribution to the error of every weight in the network can be determined and in the next step the weights are adjusted in order to lower the error.

2.3.5 Recurrent neural networks

The recurrent neural networks (RNNs) are a special case of NNs extensively researched beginning in the 1990s. Feedforward NNs are meant for static tasks, where all instance features are presented to the model simultaneously and there is no explicitly stated temporal dynamics in the data. On the other hand, RNNs allow the input to be presented sequentially, simulating the temporal dimension flow. Data connected to each time step (time is usually considered to be a discrete quantity for the purpose of RNNs, as a contiguous deception of time is difficult to use with them according to [50]) is inputted in the chronological order and the model is expected to utilize the temporal patterns.

The first generation of RNNs evolved from feedforward NNs. The key idea was to add previously unused connections – either between neurons in the same layer (a special case is a self-loop on a neuron) or between neurons in different layers, but directed in the opposite direction (called feedback connections [51]). The result was that cyclic structures appeared in the network, and the model was thus able to memorize pieces of information computed in the previous steps and to use them later.

While these models gained the ability to use the outputs of neurons from previous steps and could utilize the temporal dimension, the memorization ability was seriously limited. The *exploding* and *vanishing gradients* problems (introduced in [52]) prevent these neural networks from utilizing long-time dependencies as the importance of chronologically distant observation can either grow or vanish exponentially fast with time. Such observation was therefore impossible to utilize with this structure.

2.3.5.1 Long short-term memory NN

A solution to these problems was found in 1997 when Sepp Hochreiter and Jurgen Schmidhuber proposed an innovative technique called *long short-term memory* unit in [53]. This unit was meant to replace the standard artificial neuron and to solve the vanishing and exploding gradient problems. It has more complicated structure than neuron, with an inner stored state and three important parts (called *gates*):

- **Forget gate** decides what part of inner state should be kept and which pieces of information should be forgotten.
- **Input gate** decides which part of the input should be added to the internal state.

- **Output gate** decides which part of the inner state (already updated by the other two gates in this step) should contribute to the output value.

Each of these gates has its own parameters, which have to be trained before the usage. The inner structure is shown in figure 2.2. According to [53], LSTM networks are capable of processing dependencies longer than 1000 time steps, which is much more than other models. It has been proven that LSTM NNs are capable of solving very difficult tasks and nowadays they are successfully utilized in many fields (for example Graves proved in his 2013 study of speech recognition that LSTM NNs outperform any other model [54]).

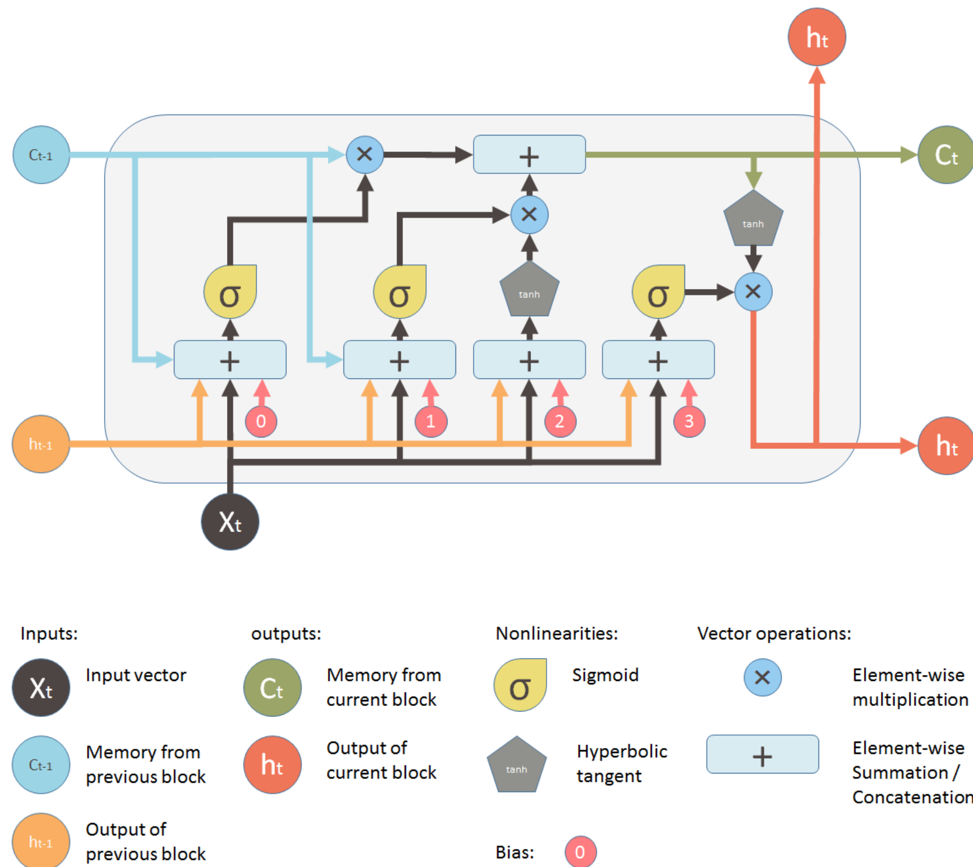

Figure 2.2: The inner structure of a LSTM unit (taken from [55])

2.3.5.2 Gate recurrent unit NN

A variant of LSTM unit called *Gated recurrent unit* (GRU) was introduced in 2014 by Cho et al. [56]. Its inner structure is simpler compared to the LSTM unit, with fewer parameters and completely missing output gate. Despite

this, it is still comparable to LSTM NNs in most tasks. There are also many tasks where the simplicity of GRU allows easier training of models and better performance compared to LSTM NNs, especially in tasks where only smaller datasets are available [57].

2.3.5.3 Stacked LSTM NN

LSTM and GRU NNs are typically constructed by several corresponding units in one layer. Their depth comes not from the count of layers (unlike feedforward NNs) but rather from their recurrent nature and the number of time steps. Despite this, in 2013 Pascanu et al. proposed several ways of connecting RNNs into the multiple layer structure [58]. Outputs of each layer are provided to another, which resembles the feedforward NN structure (although there are usually significantly less layers because of the extensive computational cost of LSTM NNs – the original paper experimented with only two layers). This approach was named Stacked RNNs and situationally can outperform other models.

Chapter 3

Time series prediction experiments

The ultimate goal of experiments described in this chapter is to create an accurate way of predicting whether a user is interested in listening to music at a certain moment of time. Recommending music at the right time might be really important for the success of a streaming service. A good example would be a mobile application that can suggest a song at any time, even when the user is currently not listening to anything. Showing notifications with suggested songs may result in a higher usage of the application since many users might find this feature comfortable. However, if such notifications are shown at an inappropriate time, users might be dissatisfied with the application and there is a huge risk of losing customers.

3.1 Summary of used data

Dataset used throughout this chapter was provided by the Recombee [6] company. It contains records of music listening patterns of over 345 000 real users, collected from an unspecified music-streaming service. Each user has a history of playbacks gathered from 30 July 2016 to 23 February 2017 (approximately 200 days). Tracked records are of one of two types:

- **Finished playback** is a playback of individual song that the user listened to its end. It contains timestamp of the start of playback and a short description of the song consisting of unique numerical identifier, name, performing artists and genre.

- **Skipped (unfinished) playback** is a playback of individual song that the user decided not to listen to its end. Generally that means the user skipped this song or ended his listening session. It contains timestamp

[6] www.recombee.com

of the start of playback and a short description of the song consisting of unique numerical identifier, name, performing artists and genre. It does not contain the duration for which the user were listening to this song.

The distribution of playbacks can be seen in figure 3.1. There is an additional category (called *repeated*) for depicting playbacks which were unfinished because the user chose to repeat the song in a certain moment. Such playbacks were labeled in the dataset as *skipped* but in fact they make up their own category and can be considered as a sign that the user enjoyed such song.

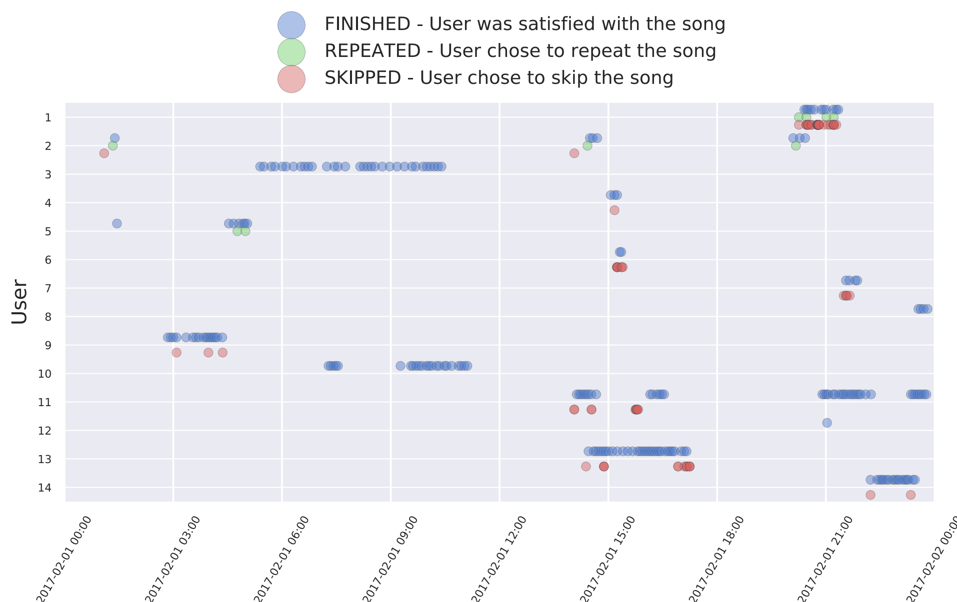

Figure 3.1: Distribution of playbacks of 14 random users during one day

Although the number of users in this dataset is huge, majority of them did not use the service for long enough to provide a sufficient amount of data. Because of that, the decision for the following experiments was to select only a set of the most active users. The main criterion for this selection was the total number of playbacks (not taking into account whether finished or skipped). Only users with more than 1 000 playbacks and less than 5 000 playbacks were included in this set. As the playbacks were recorded in the span of approximately 200 days, it can be roughly thought of as a set of users who played 5 to 25 songs per day on average (or even more for users who were not active the whole period). The reason why these limitations were chosen is simple. Less active users have the average of only a few records per day at most which makes it much more difficult to discover patterns in the data. As shown later in this chapter, even the data of the most active users are sparse with a high level of noise. The upper bound for number of playbacks

is meant to exclude a small amount of outliers in the data. Playing more than 5 000 songs signifies either a really untypical user who listens to music far more than the rest of the population or potential abuse of the service, for example continuous playback of music in public places like stores. Either way, including those users would introduce data that are possibly biased in an undesired manner. Only 64 outliers (0.02 % of the dataset) were removed for having too much playbacks, yet those had over 3 % of total playbacks. The selected set of most active users contains 1 810 users. Comparison of the whole provided dataset and selected set is shown in table 3.1. In the rest of this chapter, only this selected set is used and may be referred as *the dataset* or similarly.

Table 3.1: Comparison of the whole dataset and the selection of most active users

	Whole dataset	Selected users
Number of users	345 234	1 810
Total number of playbacks	17 547 599	3 085 899
Average number of playbacks	51	1 705
Median of playbacks	10	1 430
Percentage of finished playbacks	72.6 %	72.3 %

3.2 Data preprocessing

3.2.1 Discretization of time

In order to transform the problem of predicting user behavior into a time series prediction task, it is necessary to change the format of the input. Initially the data for each user were in the form of playback records with an attached timestamp denoting the starting time using seconds as the lowest resolution unit. A much more suitable representation of input for experiments in this chapter is using a categorical variable denoting time instead of a numerical one. Through the process of *discretization* (often called *binning*), time is transformed into series of categories, each containing all playbacks that happened during a short period of time. A uniform value of 15 minutes was chosen as a suitable duration for each time period, which results in an amount of 96 time bins per day and approximately 20 000 time bins per the span of the whole dataset. Usage of such value preserves most of information but provides a good level of generalization. Most of the songs in the dataset are labeled as *pop* or *rock* songs, which implies the length of a typical song in the dataset is three to five minutes. Therefore the value of 15 minutes per bin is high enough to not affect the data in a harmful way. For example when there is a bin without any user activity, it is safe to assume that the user did not listen

to anything during that period, rather than that he listened to a song with the length of 20 minutes starting near the end of previous period.

Time expressed as a categorical valuable is more suitable for various models, especially those related to recurrent neural networks which usually expect uniform time step that separates following observations. Discretization is also often used to reduce the amount of noise in the data, which might be another benefit of this preprocessing.

3.2.2 Binning of activity

It was also chosen to apply binning operation to the measures of activity. After time discretization each user had a series of time bins, each having three variables – number of finished, skipped and repeated playbacks. These were replaced by one variable with only two possible values. Either there was at least one record of activity in a particular time bin (regardless whether finished, skipped or repeated playback) and user is thus considered to be active in this time bin, or there was none and user is considered inactive. An idea behind this generalization is that the amount of recorded playbacks made by user in a particular time bin is highly dependent on various circumstances as song length, temporal distractions of users or the quality of suggestions provided by the service used to record this dataset. All of these can be seen as noise. Therefore in the following experiments, two users of whom one skips 10 playbacks and repeats his favorite part of one song few times and the second one who listens to two long songs with a short break between them are considered equally active during this time interval, even though the first generates much more records of activity in the dataset. Throughout the experiments (and in source codes), this variable is recorded as a binary value where 1 stands for activity and 0 for inactivity of a user. The preprocessing done so far is shown in figure 3.2.

3.2.3 Transformation into a supervised ML task

Supervised learning tasks require a format of data consisting of pairs of feature vectors – one containing input features and the second containing output features. Such pair is called *sample*. Training and test sets contain multiple samples. To transform the user activity sequence into this format, initially a simple *rolling window method* was used to extract a vector of lag features (observations from the past) as inputs and the one following observation as output. This simulates a basic situation where in each moment a sequence of past observations is available and the task is to predict the activity value for the following time step in the future. This process is depicted in figure 3.3.

However, this proved to be an unsatisfactory way of preparing the data. A preliminary series of experiments (which are omitted from this text) using various models showed that this approach leads to results that are very close

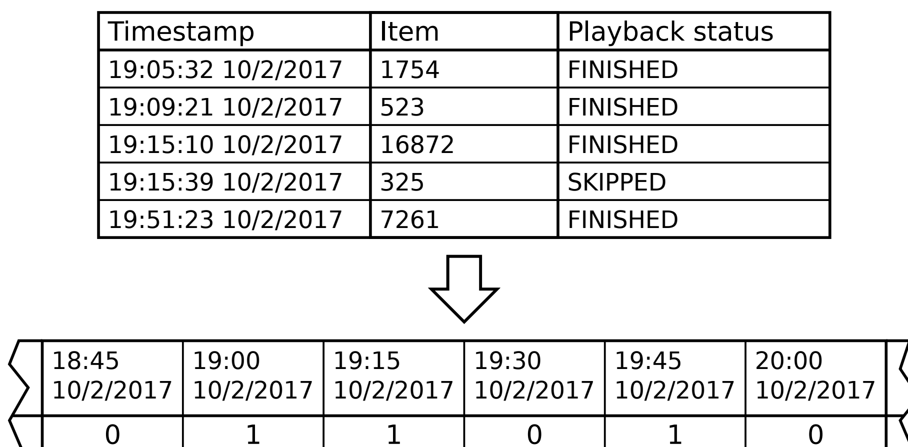

Figure 3.2: Binning preprocessing of raw data into an activity sequence

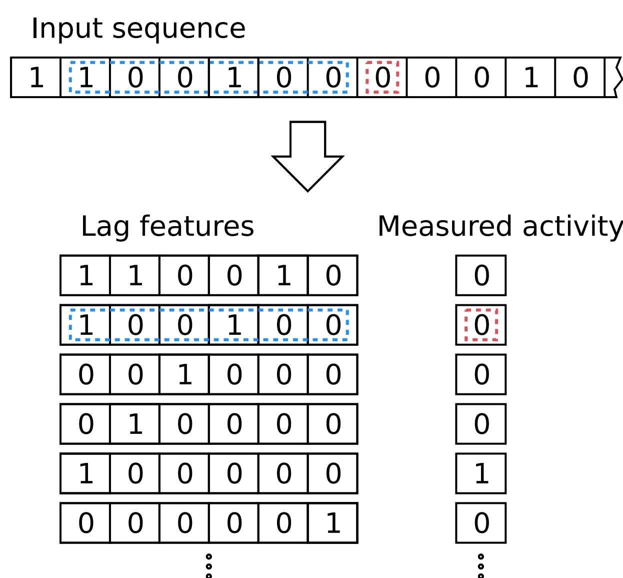

Figure 3.3: Initial extraction of samples

to the *persistence forecast model*. This is a simple time series prediction model that always predicts the last observed value, or a value highly influenced by it (a further explanation and example of persistence forecast model can be found at [59]). It is indeed a very successful model in terms of accuracy, because two consecutive activity values in this task are strongly correlated – users usually listens to music continually for multiple subsequent time intervals before becoming inactive for a prolonged period of time. The problem is

that this model's behavior is absolutely undesirable from a business point of view. A primary goal of this thesis is to predict moments when the user is in a good mood to listen to music, but is not currently listening. None such situation would be predicted if a last activity value was predicted every time. Persistence forecast model also predicts that active user would stay active forever, which is a major flaw. An example of such observed predicting behavior is shown in figure 3.4

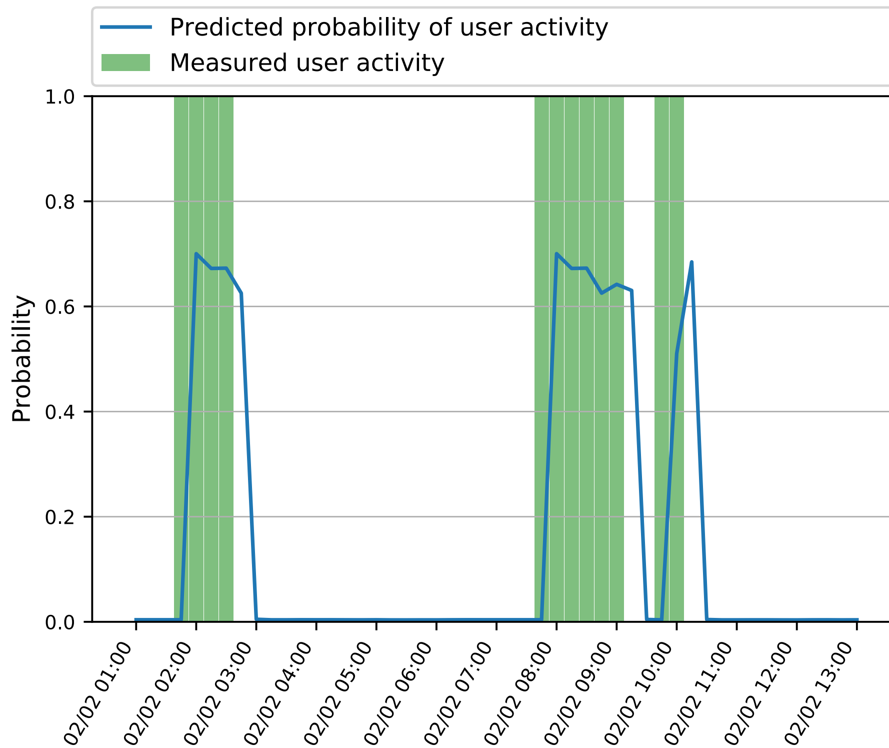

Figure 3.4: Example of persistence forecast prediction behavior

To solve this issue, two changes in data preprocessing were made. The first one is related to the distribution of activity values. An important observation is that users are mostly inactive. An average user in the dataset is active only 2 % of time. This issue is known as the *class imbalance problem* and it is a common issue when working with real-world datasets in the field of machine learning (its description and possible solutions can be found in [60, 61]).

A trivial solution to this problem is to apply *sample weighting* (in certain sources called *sampling*) to increase the importance of the less frequent class. Samples of this class are given positive weights larger than one (which is a default weight for samples) and the loss function during model training and testing is computed with emphasis on these samples because their prediction

error is multiplied by their weights. This approach (called *oversampling*) would be most likely functional, but a slight alteration is used in this thesis. Instead of assigning each sample where the expected output is “active” a larger weight, it was done so only in cases where the last (most recent) observed lag feature is “inactive”. That is because the main goal is to predict changes in user activity from the value of 0 to 1 (e.g. when inactive user becomes active). Samples in which the last observed lag feature is “active” remain weighted by the default value of 1. A weight for samples, which has their weight increased, can be compute in the folowing way:

- First, a value *inactivity period length* is computed. It is the count of consecutive 0 values in the lag features vector, which are followed by no value 1 in this vector (eg. length of part of the sequence which ends at the end of lag feature vector where only 0 values are present).
- A sample weight is then computed using the following function.

$$f(inactivity_period_length) = a \log_2(inactivity_period_length) + b \quad (3.1)$$

The purpose of that is to increase the importance of samples describing situations in which user becomes active after a longer period of time. Correctly predicting that the user would become active after three days of not listening to music is clearly more desirable for potential business usage than predicting that the user would return after a 15-minutes break. This computation uses logarithm function, to avoid assigning undesirably high weights in situations in which user becomes active after an extensively long period of time (eg. several months). The values of constants a and b were chosen experimentally as $a = 3$ and $b = 5$, which ensures that the total importance of both output classes is approximately the same for users within this dataset. This approach is depicted by green markers in figure 3.5.

The second change done to the data format is altering the output feature. Previously it was just the measured value of the following time step. Preliminary experiments showed that a better choice for the output feature is a more generalized value, computed as a maximum of values of the 4 following time steps. This can be seen as a value signifying whether a user is predicted to become active during the following hour (as one time step is 15 minutes). This has three advantages:

- It reduces the level of noise as there are fewer changes caused by temporal inactivity periods of users.
- It slightly increases the number of samples with output feature of value 1, which in addition to the sample weighting solves the class imbalance problem.
- It predicts the user activity “earlier” – there is usually a larger gap between the moment a model predicts user activity and the moment the

user becomes active. This is the most important part. The ability to forecast user activity in the following hour instead of 15 minutes might be beneficial for usage in real-world recommender systems. Predictions could be potentially computed less frequently and during the larger gap there would be more time for additional tasks like running a collaborative filtering model to predict an item that would be afterwards suggested to the user. This is however just a speculation without knowing the structure of the recommender system.

The overall impact on resulting predictions is that they are more robust and useful. This change is also shown in figure 3.5.

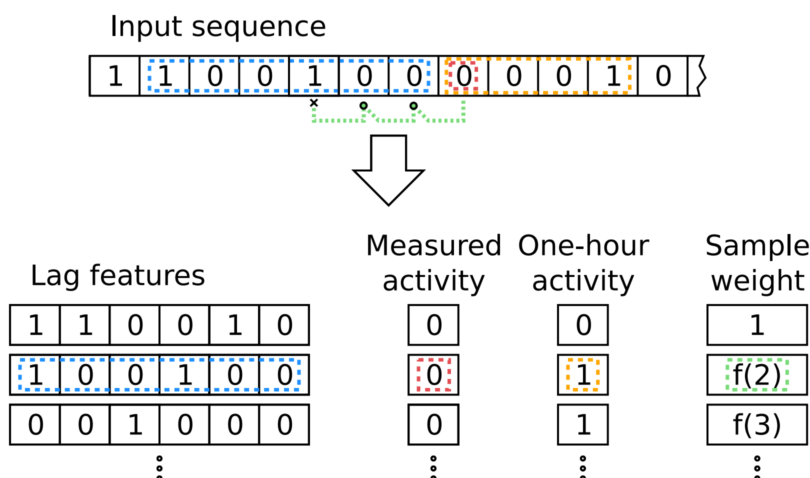

Figure 3.5: Advanced extraction of samples

3.3 Structure of experiments

3.3.1 Overall structure

This section describes shared aspects of all experiments conducted in this chapter. The main thing to notice is that a different model is constructed, trained and evaluated for each user, using only the collected data of this user. This is a purely academical approach, which cannot be used in real production environment because of the extensive computational and memory costs for having such a high number of models. This issue is addressed in the next chapter where shared models are used for predicting multiple users.

To gather comparable results from various different models, a common schema of providing data to them is used. For each user (and corresponding model) the preprocessed samples are divided into two sets by their chronological order – the first set contains all samples that hold information about

actions before a chosen threshold date and the second contains the rest. The first set is available to the model in the training phase while the second is hidden during training later used for evaluation of the model. The testing set contains 3 996 samples (20 % of the total count), which corresponds to testing models over a period of approximately 42 days of simulated usage.

The models predict future activity in a *one step ahead* manner, which means that they have knowledge of all previous time steps when predicting the next one.

3.3.2 Model behavior

The task of predicting whether a user is expected to be active or inactive can be easily considered a classification task (with two classes). However, such solution might not be the best from a business point of view. A strict division into two classes can be harder to utilize as it does not reflect the level of uncertainty of predictions. Because of that, regression models are used to solve this task. A value predicted for every step is an expected probability that a user will be active. That means this value is a real number from range $[0, 1]$. Certain used models can naturally produce a value higher or lower. Because of that, outputs of these models are normalized by using sigmoid function with range $[0, 1]$. An important advantage of regression approach is that the resulting activity predictions can be subsequently tuned and utilized in numerous ways, for example by adjusting the threshold for selecting intervals in which the user is expected to be active.

3.3.3 Measuring results

To measure an overall accuracy of models, ROC curve is used. The whole curves can be compared, but most of the time their AUC is a sufficient metric. If multiple results are evaluated and their results have to be added together (for example when evaluating multiple models in one experiment), an area preserving algorithm is used (as it is described in [62] in section *Non-parametric methods for receiver operating characteristic curve averaging*). The use of ROC explains why there is no loss in prediction accuracy when normalizing outputs to fixed range. Sigmoid function is monotonic and thus using it does not affect ROC curve.

Although ROC AUC is a reliable metric, it is not convenient to use it for actions like model validation or as a loss function when training models such as neural networks, mostly due to its computational cost and complexity. Because of this, MSE (mean square error) or RMSE (root mean square error) is used when training and validating models. These metrics are used as a *surrogate* metrics (simpler metrics providing accuracy approximation).

3.3.4 Further data selection

The only issue with using ROC AUC in this task is the consequences of dealing with class imbalance in the original data. ROC curve that is computed when not enough samples of one class are available can be highly inaccurate (with AUC close to 0 or 1). In the dataset, there are number of users that were active only several times or even never during the testing time interval. To obtain more meaningful results, it was chosen to exclude all users that were active during less than 60 time bins in this interval (from the total of 3 996, i.e. 1.5 %, while the average ratio of active bins amongst users is 2 %).

The resulting count of users is 946. This number can be considered to be fully sufficient for the purpose of this thesis. As it is a good practice in machine learning not to perform model selection and optimization on the same data that are used for final results evaluation, these users were divided into two disjunctive sets:

- Set of 757 users (80 %) that are used for model training (especially for multi-user models discussed in the next chapter) and for model selection and hyperparameter optimization of such models
- Set of 189 users (20 %) that are used solely for evaluation of selected models

3.3.5 Implementation and environment

All models and experiments were implemented in the programming language Python (version 3.5). The most important package used in this thesis is the Keras framework [63]. It is a modular library used for the construction of neural network models implementing a variety of deep learning structures and algorithms. Keras uses Tensorflow framework [64] as a backend for computations. All neural network models used in this thesis were built in Keras. For storing the original dataset and additional preprocessed data, PostgreSQL 9.5.12 was used.

Experiments were run on a mid-range notebook having Intel Core i7 CPU (4 cores, 2.2 GHz), 8 GB RAM and Linux Mint 18.1 operating system. This proved to be a limiting factor during several computationally difficult experiments. Especially hyperparameter optimization of complex neural network models was conducted only in a simplified way. There is no doubt that better results can be achieved, should the experiments be repeated using more resources.

3.4 Used models

3.4.1 Baseline statistical models

This set of simple models was created to explore some basic data patterns and to provide baseline results.

The first model (later in this thesis called *Time of day model*) is a very trivial one. It groups all action bins by their time of day (thus there are 96 groups) and then calculates the ratio of those in which the user was active for every such group. The prediction is then the ratio for the appropriate time of day. This model utilizes the assumption that the actions of users are periodical in the timespan of 24 hours and a particular user is likely to be active at the same time of day as he was in the past. Although this model is simple and requires no expensive training, it performs rather well with an ROC AUC value of 0.747 on test data (0.775 on training data). This is a strong indication that time of day is an important feature for predictions.

The second model is similar, but it groups action bins by the corresponding day of the week (there are 7 groups). With an AUC of 0.568 on test data (0.590 on training data) it is worse than the first model, but still better than an uninformed (random) classifier which means user actions are also slightly periodical in the timespan of a week.

The last model is the combination of the previous two. Both the time of day and day of the week are taken into consideration (which corresponds to 672 groups). The resulting AUC is 0.731 on testing data (0.861 on training data), which is worse than the first model – the higher number of groups do not allow the model to generalize so well. The resulting ROC curves are compared in figure 3.6.

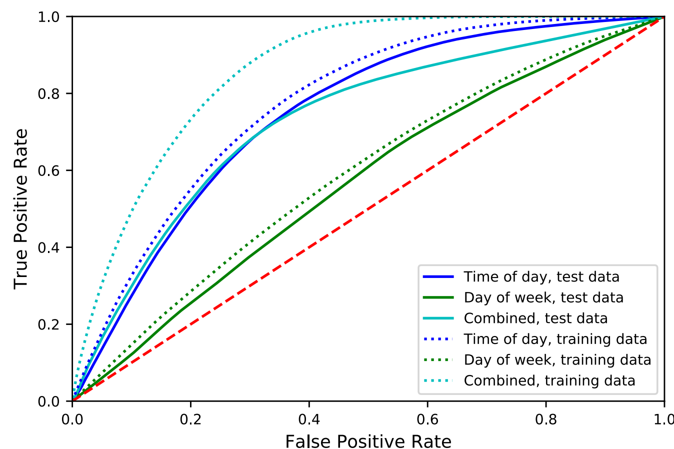

Figure 3.6: Comparison of simple baseline models

3.4.2 Perceptron model

Another simple model is a single perceptron. It is the most trivial example of neural network with no hidden layers and a very limited expressive power. It behaves as a linear regressor, meaning that the output is a linear combination of input features. A validation set containing randomly chosen 33 % of samples is excluded from training data. Validation using these data helps to stop the training process as soon as no significant improvement is measured on validation set.

The resulting ROC AUC of 0.718 on test data (0.882 on training data) is nothing remarkable. A more interesting output of this model is the feature importance, which can be determined by measuring the absolute value of weights trained for each input feature [7]. This value signifies how important each input feature is for the resulting output. The results (generated by averaging the values gathered over the set of testing users) are shown in figure 3.7. In this figure, the features are sorted in chronological order (as is implied in the preprocessing section), where feature indexed -1 is the latest.

Feature importance gathered by a simple model can provide a convenient way to explore some basic patterns in the data. In this case it can be seen that features corresponding to lag observations from the same time of the day for which the prediction is made (depicted by dotted vertical lines) are more important. A significant increase in importance can also be observed in periodical peaks every 7 days.

3.4.3 Gradient boosted trees model

Gradient tree boosting is capable of finding patterns that linear model is unable to express and it is a popular and versatile model. However, in this particular task it did not yield good results. Both the basic and stochastic variant of gradient boosting were used with similar results. Basic variant has the ROC AUC of 0.692 on test data (0.908 on training data) and the stochastic variant has AUC of 0.693 on test data (0.900 on training data). The only difference was that the stochastic variant was almost two times faster to train.

The most likely explanation is that the preprocessing made is not suitable for gradient boosting as there are too much features, which also tend to be noisy. Both variants of gradient boosting started to overfit after approximately 500 iterations of boosting. The scikit-learn [66] implementation of this model was used.

[7] While determining feature importance from any linear regression model is straightforward, it is interesting that the same approach (i.e. measuring weights on neurons in the input layer) can be (to some extend) used for an approximation of feature importance in more complex neural network models. This is partially shown in [65].

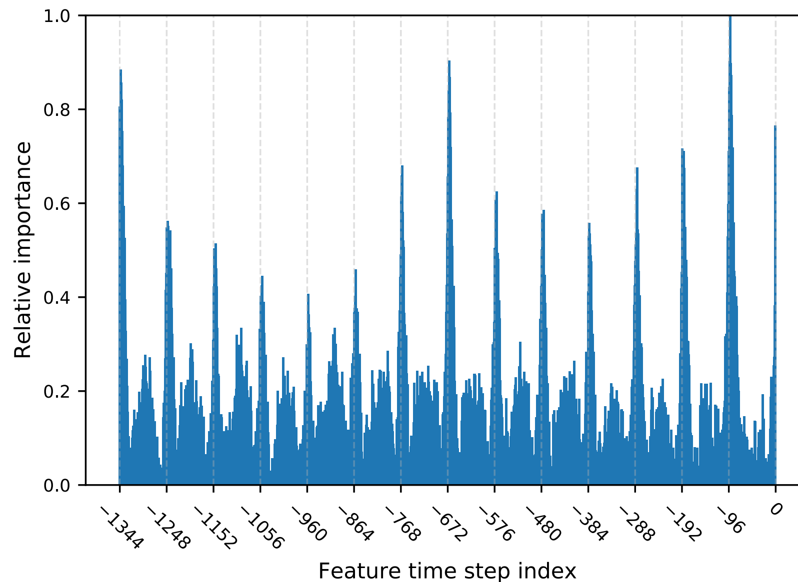

Figure 3.7: Relative importance of features

3.4.4 Multilayer perceptron model

Multilayer perceptron is a standard feedforward neural network model capable of extracting complex patterns. By a simple *grid search* a structure of 15 hidden layers each containing 50 neurons was found optimal. A 33 % of training data was used once again for validation of model and for determining the optimal stoppage point of the training process. More robust approaches like *cross validation* could not be used due to the high computational cost of training neural networks. Two variants of MLP were constructed and optimized – a basic model with no regularization method and a MLP using *dropout* as a regularization method. A dropout rate of 0.3 was found the best during by a series of experiments. Additional settings were used as advised in the original paper concerning dropout [67].

Nevertheless, MLP has proven to be an unsuitable model for this task. The basic variant accomplished an AUC of 0.683 on test data (0.884 on training data) and the dropout variant 0.654 on test data (0.849 on training data). The problem is that the MLP model quickly overfits no matter what hyperparameter optimization is done. This behavior is discussed later in this chapter.

3.4.5 LSTM model

The last model constructed in this chapter is a LSTM neural network. In theory, this model should be the most accurate as recurrent neural networks are especially successful at solving time series prediction tasks. A *stateless*

variant (inner states are randomized for each instance, in contrast to *stateful* variant) with only one layer of LSTM blocks were used. *Adam* was chosen as a stochastic optimizer during the training process (as described in its original paper [68]). By multiple hyperparameter optimization experiments, a structure of 40 LSTM blocks was found optimal. This LSTM model scored a ROC AUC of 0.631 on test data (0.914 on training data).

As the dropout regularization application on recurrent neural networks is not yet researched enough and is generally not recommended (according to [69, 70]), a standard L2 weight regularization (as explained in [71]) is used. The higher the regularization parameter the more are large weights in the internal structure of neural network penalized and the model becomes more generalizing. This can be seen in figure 3.8, which depicts results of experiments run over the set of 40 training users. Regularization clearly helps with reducing overfitting in this case. The value of 0.005 emerged as the best settings. With this regularization method, the resulting AUC is 0.678 on test data (0.865 on training data).

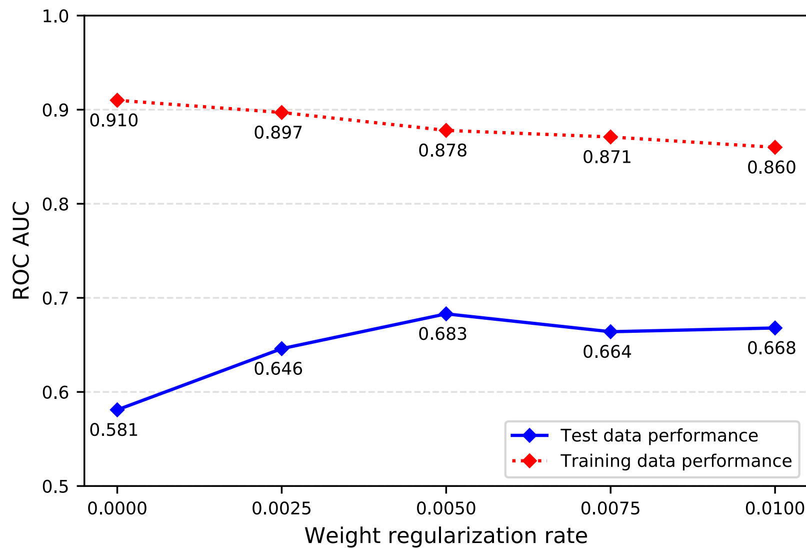

Figure 3.8: Comparison of model performance with varying regularization settings

3.5 Conclusion of results

The measured performance of all models is compared in table 3.2. The last column shows an average training time per one model. It can be observed that simple models perform better than the complex ones.

Table 3.2: Comparison of performance of basic models

Model	Test AUC	Training AUC	Training time (s)
Time of the day baseline	**0.747**	0.775	<1
Day of the week baseline	0.568	0.590	<1
Combined baseline	0.731	0.861	<1
Linear regression – perceptron	0.718	0.882	35
Gradient tree boosting	0.692	0.908	86
Gradient tree boosting – stochastic	0.693	0.900	52
Multilayer perceptron	0.683	0.884	120
Multilayer perceptron – dropout	0.654	0.849	193
LSTM	0.631	0.914	267
LSTM – weight regularization	0.678	0.865	287

This can be most probably interpreted as a problem with input data format – and indeed there are issues with it. Before adjusting the weights of samples, the models behaved too similarly to persistence forecast model and there was a severe class imbalance in the input data. Both issues were solved by the proposed variant of oversampling, but this brought another problem. A typical situation for data of one user is that amongst the total approximatively 15 000 samples in training set, there are several hundred samples, which weights are adjusted by a large factor (commonly 30–50). This is shown in table 3.3. As a consequence, complex models are likely to lose their generalization potential and to overfit, which is a well-known issue of oversampling [61]. The structure of experiments, where one model is constructed for each user, might seem optimal because each model is fully personalized. However, the number of samples labeled *active* for each user is too small, which enhances the negative effects of oversampling.

Table 3.3: Comparison of relative count and importance of samples grouped by their sample weight determined during preprocessing

Weight	Relative count	Importance
1-9	97.82 %	58.33 %
10-19	0.27 %	2.40 %
20-29	0.46 %	6.80 %
30-39	0.94 %	18.86 %
40-49	0.42 %	10.89 %
50-59	0.07 %	2.23 %
60-69	0.01 %	0.33 %
70-79	0.00 %	0.15 %

The main problem of experiments in this chapter is therefore the lack of data, which prevents from solving the described problems without making models prone to overfitting. In the next chapter a different model structure and data preprocessing are used, adjusted according to observations made in this chapter.

Chapter **4**

Advanced experiments

As previously stated, there is couple of problems with the *one user – one model* structure. An alternative way of preprocessing and model construction is proposed in this chapter to mitigate issues observed during previous experiments. The key change is that only one model is constructed and shared by all users. This lowers the personalization aspect but helps to solve the lack of data problem and the connected issue with rapid overfitting of complex models.

4.1 Data preprocessing

Most of the preprocessing remains the same as in the previous chapter. The samples of all users are prepared in an unchanged way using discretization of time, binning of activity and the same format of transformation into supervised learning task and sample weight computation. The key difference is the way in which those weights are used.

4.1.1 Selection of training data

As described in the previous chapter, there are two sets of users – training set containing 757 users and test set containing 189 users. For the purpose of training the model, data of 657 users from the training set was used (the data of the remaining 100 users was used later for hyperparameter optimization and therefore had to be excluded from this set). If all data from such 657 users had been simply added together and used, the training set would contain approximately 9.5 million samples. Not only that would require memory space of hundreds of gigabytes (in Python implementation), but the computational cost for training models would be too high. Such set of samples would furthermore once again suffer from class imbalance and the models would be prone to simulating persistence forecasting as explained in the previous chapter.

The difference in this case is that there is enough data to avoid the need to use oversampling to remove the class imbalance. A simple *undersampling* which takes sample weights into account is used instead. Undersampling in its basic form is a process of removing certain samples from the dataset. In this case, the samples to be removed are chosen randomly, but the probability of a sample to remain in the set is adjusted by its weight. For example, a sample with weight 30 is thirty-times more likely not to be removed compared to a sample with weight 1. This way the desired layout of data is achieved and all remaining samples are considered to be of equal importance. The final number of samples chosen is 198 977. This set of samples is the same for all experiments in this chapter and is used solely for training models. The process of testing the models remains unchanged.

4.2 Used models

4.2.1 Perceptron model

As a baseline model for this chapter, linear regression in the form of a single perceptron model was used. The resulting ROC AUC was 0.733 on test data (0.789 on training data). That is much better performance than what was accomplished by the same model using the *one user – one model* structure (which resulted in an AUC of 0.718 on test data and 0.882 on training data). The difference in training data AUC values promises that overfitting is less likely to happen to models with this input data structure.

4.2.2 Multilayer perceptron model

This model is much alike its counterpart in the previous chapter. The only difference in the implementation is that the hyperparameter optimization is not done by a simple *grid search* but instead by a Python package Hyperas [72]. This is a simple tool for Keras build upon the Hyperopt [73] framework. Hyperopt can be used to automatically search for the best hyperparameters of a neural network, utilizing a *Tree of Parzen Estimators* method for accelerating the search (this method is described in [74]).

First, a model with no additional regularization was constructed. Using the presented tool a structure of 6 hidden layers with 20 neurons in each was selected. This model was evaluated to have ROC AUC of 0.741 on test data (0.801 on training data). A second model used dropout as a way of regularization. Using Hyperopt, a structure of 6 hidden layers with 40 neurons in each and a dropout ratio of 0.266 was chosen. This model performed sligthly better with AUC of 0.743 on test data (0.820 on training data).

4.2.3 LSTM model

The advantage of recurrent neural networks lays in the ability to better understand sequential nature of time series prediction tasks than regular feed-forward neural networks. This expectation was confirmed as a basic LSTM model outperformed all previous models with a resulting ROC AUC of 0.750 on test data (0.819 on training data). That means that this model is the first that finally achieved better results than the first baseline model that considers only the time of day. It is interesting that to accomplish this accuracy, only 50 LSTM blocks were needed and adding more did not help to improve performance. Even with lower number of blocks the model performs very well as shown in figure 4.1 (notice that the AUC values were measured during the phase of model selection on set of validation users and are different from the value optained during testing).

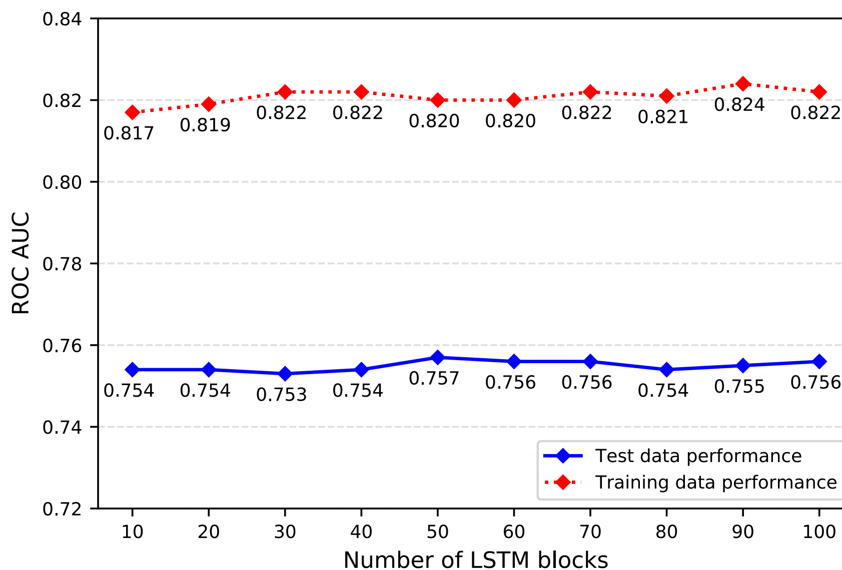

Figure 4.1: Comparison of model performance with varying number of LSTM blocks

A second model was constructed to use regularization to determine whether its addition would improve the result. Identically to LSTM model in the previous chapter, L2 regularization of weights of the model was used instead of dropout. Using Hyperas, it was determined that a structure of 61 LSTM blocks with regularization rates of 0.004 for forward connections and 0.01 for recurrent connections in the network is optimal. This model achieved an ROC AUC of 0.750 on test data (0.812 on training data), which means a performance identical to the variant without regularization.

4.2.4 Stacked LSTM model

To further elevate the pattern-descriptive ability of previous model, multiple layers of LSTM blocks were added together. Using 50 LSTM blocks in each layer, the optimal number of layers was found to be 5. More layers not only slowed down the training process, but also performed worse on both training and test data. No weight regularization was used because it was observed that it negatively affected the resulting performance. Stacked LSTM model achieved a ROC AUC of 0.754 on test data (0.822 on training data) which is a further improvement over one-layer LSTM model. This structure, which utilizes the deep learning approach of more layers, clearly allows the model to find more complex patterns in the data than the basic variant. An improvement of 0.004 in AUC might seem to be marginal but in such a task where most errors could be addressed to a noise in the data, every improvement above certain level becomes more and more difficult.

4.2.5 Stacked GRU model

A GRU variant of recurrent neural networks is often considered to be viable replacement over LSTM NNs. Because the GRU is simpler than the LSTM unit, there are many tasks that can be solved by GRU NNs with comparable or better performance and less computational time – however, this cannot be stated generally for every task. Interestingly, the GRU variant also performed best with 5 layers, each containing 50 blocks – the same inner structure as the LSTM model. Despite the expectations, Stacked GRU model performed worse than LSTM NN, with ROC AUC of 0.749 on test data (0.821 on training data). A slightly shorter training time is the only advantage of this model.

4.2.6 Ensemble model

Combining multiple models together is a commonly used strategy to overcome limits of individual models. Although the models described in this chapter perform well and have several advantages over models constructed in previous chapter, there are two critical drawbacks:

- Although every sample provided to the system contains a part of user's historical data, having one model for every user means that the predictions are not fully personalized. This is an undesired behavior as many examples from the fields of recommender systems and machine learning show how important personalization is in order to process user data accurately.

- Every sample is preprocessed to contain historical data from the last 14-day window only. Patterns spanning over longer period of time cannot be recognized. Although this limitation is reasonable for simplifying the

model construction process, the system does not utilize all data available and therefore is likely not to achieve optimal results.

On the other hand, the baseline *Time of day model* (described in section 3.4.1) is fully personalized and takes all measured data into consideration. Furthermore, it is decently accurate and also very simple. It is not able to recognize more complex patterns and so behaves in an opposite manner to more advanced models like Stacked RNNs. This makes the baseline model an ideal addition to such models. The ensemble model therefore consists of two separate models:

- Stacked LSTM or Stacked GRU model, shared by all users
- Baseline *Time of day model*, unique for every user

Both models were run at once and their output values were aggregated by a third component – the stacking model. In this case it is a rather simple feedforward neural network. Its structure consists of 5 layers with 5 neurons in each of them, making it capable of more than just a linear combination of inputs but ensuring good generalization ability through the model's simplicity. It outputs a single value with the same meaning as previous models, i.e. the predicted probability of user's activity for a single time step. It is trained by standard backpropagation algorithm using the set of validation users (which were not used during training of Stacked RNN model nor are used for testing the models) and outputs from the two already trained base models to minimize the RMSE. The structure of such ensemble model and the layout of used data during training process are depicted in figure 4.2.

The prediction quality fulfilled the expectation and was superior to all other models. Using Stacked LSTM model, the ROC AUC was 0.768 on test data [8] while the usage of Stacked GRU proved to be slightly worse with AUC of 0.766. The ensemble model is clearly better than the rest of the evaluated solutions. Its detailed description and usage analysis can be found in the next chapter. This model is proposed by this thesis as the best solution for the original task.

4.3 Conclusion of results

The overfitting issues observed in previous chapter were mitigated by performing different preprocessing and using a more general structure of one model shared by all users. A visualization of predictive behavior comparison of the structure used in the previous chapter and the *shared-model* structure used in this chapter can be found in the appendix B. This appendix also contains

[8] Notice that measuring the performance on training data makes little sense as all components of the ensemble were trained using different data.

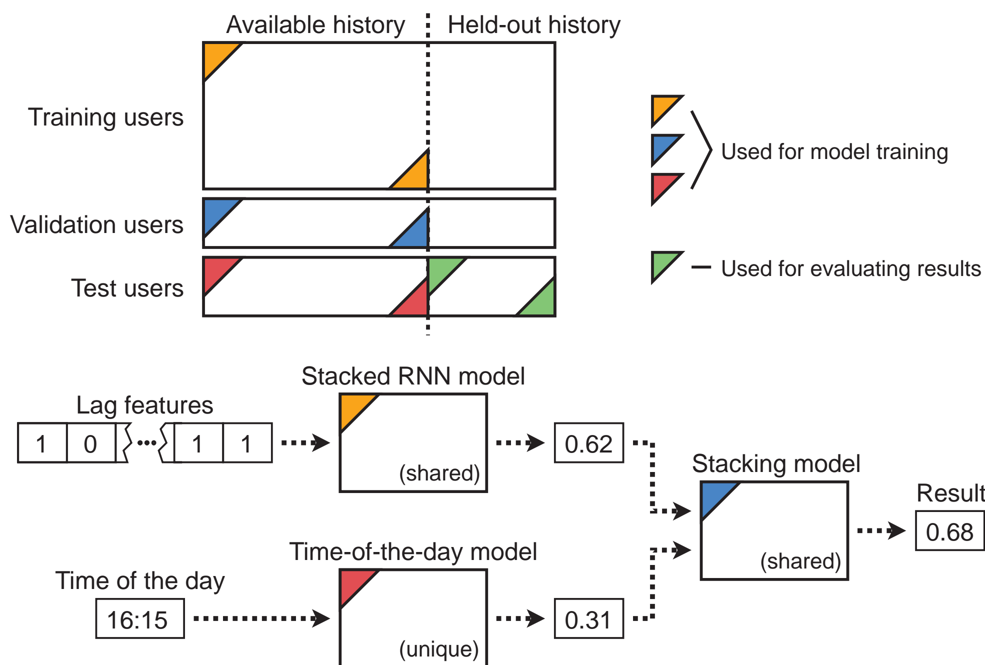

Figure 4.2: Structure of ensemble model

visualizations of predictive behavior of selected models used on a set of 5 users belonging to the test group.

The complete comparison of results of described models can be found in table 4.1. The last column shows an average training time per one model. Duration of training is intentionally omitted in cases where it depends on the number of users by design. Unsurprisingly the recurrent neural networks performed better than any other model because they are most suitable for time series tasks. By far the best results were achieved by ensemble models that use two completely different models.

Table 4.1: Comparison of performance of advanced models

Model	Test AUC	Training AUC	Training time (s)
Time of day baseline	0.747	0.775	–
Linear regression – perceptron	0.733	0.789	63
Multilayer perceptron	0.741	0.801	658
Multilayer perceptron – dropout	0.743	0.820	610
LSTM	0.750	0.819	450
LSTM – weight regularization	0.750	0.812	510
Stacked LSTM	0.754	0.822	1215
Stacked GRU	0.749	0.821	1094
Ensemble – stacked LSTM	**0.768**	–	–
Ensemble – stacked GRU	0.766	–	–

Chapter 5

Proposed solution

Although there are plenty of possible approaches to the time series prediction tasks and many more experiments could be conducted, the results observed during previous chapters clearly favor certain solutions above others. Specifically, the ensembling of different models proved to perform the best. In this chapter, this solution is thoroughly analyzed with emphasis on possible usage in real-world situations. The solution is presented in a detailed but general form with no assumption on its exact implementation and used technologies and run environment.

For the purpose of visualization of the proposed solution a very simple interactive application was made. It can be found on the attached storage medium.

5.1 Description

The main goal of the proposed system is to predict the level of activity of users in a forthcoming short period of time. The system is constructed to yield the best results according the business requirements described previously, i.e. to make predictions in a way that can be used for sending recommendations to the users.

Other aspects of this process are not solved by this thesis. The actual sending of notifications to users is dependent on the interference and should be maintained by the providing subject of such service. The system proposed in this thesis also does not explicitly declare in which moments a notification is to be sent to a user. This should be decided by other connected unit that processes the output predictions of users' level of activity created by this system. Construction and fine-tuning of such unit is out of the scope of this thesis because that can be expected to be a time-consuming task requiring online user testing to guarantee the satisfaction of future users with such service. Selecting which items should be recommended to users is also not solved by the system as it is not the goal of this thesis.

5.1.1 Structure

The system is expected to be run as a part of recommendation engine. The system on regular basis gathers data about the activity of users, processes them and returns predictions that can be utilized by the main recommendation engine. This set of computations should be done every 15 minutes as this is the time window for which predictions are made. Each time, a list of users active during the previous 15 minutes should be inputted into this system, which in return provides a prediction of activity for every user monitored by the system.

The inner structure is derived from the experiment described in section 4.2.6. A predictive ensemble model is composed of Stacked LSTM model, simple *Time of day model* for each user and a stacking model which generates the resulting values. The ensemble model's structure remains the same as shown in figure 4.2 in the previous chapter.

In order not to make the computationally difficult preprocessing from logged data each time a prediction is needed, for each user there are two data structures containing records in a format suitable for the models. The first is a list of 1 344 binary values which signify the chronologically ordered activity values measured during the preceding 14 days. These are used as lag features inputted into the Stacked LSTM part of the model. Every 15 minutes, when a new observation is gathered from the user actions, the corresponding value is added to the list and the chronologically oldest value is removed from the list. The second data structure is a table containing the number of times user was active and inactive during a specific time of day (in the form of 15-minute intervals). These values are used for computation in the *Time of day* part of the model. This table has a fixed size of 96 integer values.

5.2 Overview of usability

5.2.1 Computational and memory cost

As both the data structures mentioned previously are unique for each user, the system requires a considerable amount of storage memory. Although this is largely dependent on the implementation, an estimate is several kB per each user. That is however an amount low enough for modern enterprise environments and should not cause any issues. Other parts of the ensemble model are shared amongst all users and thus do not require significant amount of memory.

The used ensemble model is rather simple and the process of predicting is not computationally difficult. Because the activity of each user is predicted independently, the whole computation can be also easily parallelized. Huge advantage of the proposed system is the fact that the ensemble model does not need to be retrained later with additional data. Both the Stacked LSTM

part and the stacking part are trained with data containing a large number of users and thus are able to recognize the most common patterns in data. It is not reasonable to expect huge changes in all users' general behavior over time and the additional data available for training is not likely to improve the performance of the system.

5.2.2 Analysis of predictions

The fact that the proposed model performed better than any other that were examined in the previous chapters is evident from the measured values of ROC AUC. However, much more important is the usability of predictions in real-world situations. This section analyses the behavior of the system under conditions and aspects that may occur during usage.

A key ability of this system is the robustness of predictions. The parts of the ensemble model shared by all users are trained using data of large number of users which means the predicted behavior is largely generalized. The model might be slightly unprecise in predicting the activity of untypical users, but it is unlikely that the system would produce unnatural and outlying predictions, which might severely impact the satisfaction of users.

The system is able to predict the activity of new coming users as well. The Stacked LSTM part can make predictions following general patterns even with a small number of data gathered for a particular user. It might be needed to temporarily lower the importance of *Time of day* part of the ensemble until enough data is gathered to avoid overfitting, but this should not significantly affect the accuracy of predictions.

It is also worth noting that although there is a part of the ensemble model shared by all users, there is no problem caused by users living in different time zones. The format in which the data is preprocessed as an input to the model secures that this cannot impact the predictions.

5.3 Potential improvements

The system is capable of making robust and reliable predictions and is suitable for usage as a part of a commercial recommender system. However, there is a lot of room for improvement, especially concerning the accuracy of predictions. There are limitless possible approaches to solving such prediction task and this thesis explores only a small part of them. Changes in data preprocessing or model construction might bring significant improvements and enhance the usability of the solution.

A particular improvement can be reasonably expected by grouping the users by their behavior and constructing a single Stacked RNN model for each group instead of having only one model shared by all users. This would furthermore increase the personalization of the predictions and probably improve

the accuracy while predicting the activity of untypical users. Yet, such structural change is not trivial as there is a lot of research to be done to correctly decide in which way the users are to be grouped and how to adjust the model construction for this change.

Conclusion

In this thesis, state of the art of several domains was reviewed. The first researched topic was the field of recommender systems with a special focus on music recommendations and its specific aspects. Next, techniques of collaborative filtering were thoroughly described. The last theoretical review was connected to machine learning and especially to modern recurrent neural networks and their ability to predict time series tasks.

In the experimental sections, the total of 19 model variations were compared with two different approaches to the overall structure and preprocessing. Series of experiments led to the ensemble model utilizing Stacked LSTM NN as one of its base learners. This model was proved not only to be superior in terms of prediction performance, but also to be most likely fully functional given the possible restricting in real-world usage.

The behavior of this model was analyzed and potential future challenges pointed out. This means the goals of this thesis were completed and its results can be utilized by various subjects.

Bibliography

1. SPOTIFY. *Company info* [online]. 2018 [visited on 2018-05-08]. Available from: https://newsroom.spotify.com/companyinfo/.
2. MELVILLE, Prem; SINDHWANI, Vikas. Recommender Systems. In: *Encyclopedia of Machine Learning and Data Mining*. Boston, MA: Springer US, 2017, pp. 1056–1066. ISBN 978-1-4899-7687-1.
3. SCHAFER, J. Ben; KONSTAN, Joseph; RIEDL, John. Recommender Systems in e-Commerce. In: *Proceedings of the 1st ACM Conference on Electronic Commerce*. Denver, Colorado, USA: ACM, 1999, pp. 158–166. ISBN 1-58113-176-3.
4. RICCI, Francesco; ROKACH, Lior; SHAPIRA, Bracha. *Recommender Systems Handbook*. 2nd ed. New York, NY, USA: Springer US, 2015. ISBN 978-1-4899-7637-6.
5. AMATRIAIN, Xavier; BASILICO, Justin. *Netflix Recommendations: Beyond the 5 stars (Part 1)* [online]. 2012 [visited on 2018-02-21]. Available from: https://medium.com/netflix-techblog/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429.
6. KORDÍK, Pavel. *Personalized push notifications enabled by artificial intelligence* [online]. 2018 [visited on 2018-05-07]. Available from: https://medium.com/recombee-blog/personalized-push-notifications-enabled-by-artificial-intelligence-8ac057bc97ba.
7. ADOMAVICIUS, G.; TUZHILIN, A. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*. 2005, vol. 17, no. 6, pp. 734–749. ISSN 1041-4347.
8. LOPS, Pasquale; GEMMIS, Marco de; SEMERARO, Giovanni. Content-based Recommender Systems: State of the Art and Trends. In: *Recommender Systems Handbook*. Boston, MA: Springer US, 2011, pp. 73–105. ISBN 978-0-387-85820-3.

9. DE GEMMIS, Marco et al. Semantics-Aware Content-Based Recommender Systems. In: *Recommender Systems Handbook*. Boston, MA: Springer US, 2015, pp. 119–159. ISBN 978-1-4899-7637-6.
10. AGGARWAL, Charu C. *Recommender Systems: The Textbook*. Cham: Springer International Publishing, 2016. ISBN 978-3-319-29659-3.
11. FENU, G.; PAU, P. L. Modeling user interactions for conversion rate prediction in M-Commerce. In: *2015 IEEE Symposium on Computers and Communication (ISCC)*. 2015, pp. 309–314.
12. JANNACH, Dietmar; RESNICK, Paul; TUZHILIN, Alexander; ZANKER, Markus. Recommender Systems – Beyond Matrix Completion. *Commun. ACM*. 2016, vol. 59, no. 11, pp. 94–102. ISSN 0001-0782.
13. BEEL, Joeran; LANGER, Stefan. A Comparison of Offline Evaluations, Online Evaluations, and User Studies in the Context of Research-Paper Recommender Systems. In: *Research and Advanced Technology for Digital Libraries*. Cham: Springer International Publishing, 2015, pp. 153–168. ISBN 978-3-319-24592-8.
14. KOHAVI, Ron; LONGBOTHAM, Roger. Online Controlled Experiments and A/B Testing. In: *Encyclopedia of Machine Learning and Data Mining*. Boston, MA: Springer US, 2017, pp. 922–929. ISBN 978-1-4899-7687-1.
15. AMATRIAIN, Xavier; BASILICO, Justin. *Netflix Recommendations: Beyond the 5 stars (Part 2)* [online]. 2012 [visited on 2018-02-21]. Available from: `https://medium.com/netflix-techblog/netflix-recommendations-beyond-the-5-stars-part-2-d9b96aa399f5`.
16. SHI, Yue; LARSON, Martha; HANJALIC, Alan. Collaborative Filtering Beyond the User-Item Matrix: A Survey of the State of the Art and Future Challenges. *ACM Comput. Surv.* 2014, vol. 47, no. 1, pp. 3:1–3:45. ISSN 0360-0300.
17. JENSEN, Jesper Højvang. *Feature Extraction for Music Information Retrieval*. 2009. Available also from: `http://vbn.aau.dk/files/19151090/thesis.pdf`. Ph.D. thesis. Aalborg University.
18. SCHEDL, Markus et al. Music Recommender Systems. In: *Recommender Systems Handbook*. Boston, MA: Springer US, 2015, pp. 453–492. ISBN 978-1-4899-7637-6.
19. PANDORA MEDIA, Inc. *About The Music Genome Project* [online]. 2018 [visited on 2018-05-09]. Available from: `https://www.pandora.com/about/mgp`.

20. DUNN, Jeff. *Pandora has gained 5 million users in the past 3 years, while Spotify has gained 100 million* [online]. 2017 [visited on 2018-05-09]. Available from: `http://www.businessinsider.com/pandora-vs-spotify-total-subscribers-chart-2017-6`.

21. CIOCCA, Sophia. *How Does Spotify Know You So Well?* [online]. 2017 [visited on 2018-05-09]. Available from: `https://medium.com/s/story/spotifys-discover-weekly-how-machine-learning-finds-your-new-music-19a41ab76efe`.

22. JAWAHEER, Gawesh; SZOMSZOR, Martin; KOSTKOVA, Patty. Comparison of Implicit and Explicit Feedback from an Online Music Recommendation Service. In: *Proceedings of the 1st International Workshop on Information Heterogeneity and Fusion in Recommender Systems*. Barcelona, Spain: ACM, 2010, pp. 47–51. ISBN 978-1-4503-0407-8.

23. JAWAHEER, Gawesh; SZOMSZOR, Martin; KOSTKOVA, Patty. Characterisation of Explicit Feedback in an Online Music Recommendation Service. In: *Proceedings of the Fourth ACM Conference on Recommender Systems*. Barcelona, Spain: ACM, pp. 317–320. ISBN 978-1-60558-906-0.

24. AMATRIAIN, Xavier; PUJOL, Josep M.; OLIVER, Nuria. I Like It... I Like It Not: Evaluating User Ratings Noise in Recommender Systems. In: *User Modeling, Adaptation, and Personalization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 247–258. ISBN 978-3-642-02247-0.

25. OARD, Douglas; KIM, Jinmook. Implicit Feedback for Recommender Systems. In: *Proceedings of the AAAI Workshop on Recommender Systems*. 1998, pp. 81–83.

26. PAN, Rong et al. One-Class Collaborative Filtering. In: *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 502–511. ISBN 978-0-7695-3502-9.

27. LERCHE, Lukas. *Using Implicit Feedback for Recommender Systems: Characteristics, Applications, and Challenges*. 2016. Available also from: `https://eldorado.tu-dortmund.de/bitstream/2003/35775/1/Dissertation_Lerche.pdf`. Ph.D. thesis. Technischen Universität Dortmund.

28. KORDUMOVA, Suzana et al. Personalized Implicit Learning in a Music Recommender System. In: *User Modeling, Adaptation, and Personalization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 351–362. ISBN 978-3-642-13470-8.

29. LEE, Dongjoo et al. Exploiting Contextual Information from Event Logs for Personalized Recommendation. In: *Computer and Information Science 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 121–139. ISBN 978-3-642-15405-8.

30. SARWAR, Badrul; KARYPIS, George; KONSTAN, Joseph; RIEDL, John. Item-based Collaborative Filtering Recommendation Algorithms. In: *Proceedings of the 10th International Conference on World Wide Web*. Hong Kong, Hong Kong: ACM, 2001, pp. 285–295. ISBN 1-58113-348-0.

31. SU, Xiaoyuan; KHOSHGOFTAAR, Taghi M. A Survey of Collaborative Filtering Techniques. *Adv. in Artif. Intell.* 2009, vol. 2009, pp. 4:2–4:2. ISSN 1687-7470.

32. GOLUB, Gene H; REINSCH, Christian. Singular value decomposition and least squares solutions. *Numerische mathematik*. 1970, vol. 14, no. 5, pp. 403–420.

33. SARWAR, Badrul; KARYPIS, George; KONSTAN, Joseph; RIEDL, John. Incremental singular value decomposition algorithms for highly scalable recommender systems. In: *Fifth International Conference on Computer and Information Science*. 2002, pp. 27–28.

34. KOREN, Yehuda; BELL, Robert. Advances in Collaborative Filtering. In: *Recommender Systems Handbook*. Boston, MA: Springer US, 2015, pp. 77–118. ISBN 978-1-4899-7637-6.

35. SYMEONIDIS, Panagiotis; ZIOUPOS, Andreas. *Matrix and Tensor Factorization Techniques for Recommender Systems*. Springer, 2016. Springer Briefs in Computer Science. ISBN 978-3-319-41357-0.

36. KOREN, Y.; BELL, R.; VOLINSKY, C. Matrix Factorization Techniques for Recommender Systems. *Computer*. 2009, vol. 42, no. 8, pp. 30–37. ISSN 0018-9162.

37. CARBONELL, Jaime G.; MICHALSKI, Ryszard S.; MITCHELL, Tom M. *Machine Learning: An Artificial Intelligence Approach*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983. ISBN 978-3-662-12405-5.

38. SILVER, David et al. Mastering the game of Go without human knowledge. *Nature*. 2017, vol. 550, pp. 354–.

39. BRADLEY, Andrew P. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*. 1997, vol. 30, no. 7, pp. 1145–1159. ISSN 0031-3203.

40. CHAI, T.; DRAXLER, R. R. Root mean square error (RMSE) or mean absolute error (MAE)? *Geoscientific Model Development Discussions*. 2014, vol. 7, pp. 1525–1534.

41. LEWIS, David D. Naive (Bayes) at forty: The independence assumption in information retrieval. In: *Machine Learning: ECML-98*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 4–15. ISBN 978-3-540-69781-7.

42. NETFLIX, Inc. *Netflix prize* [online]. 2009 [visited on 2018-05-17]. Available from: `https://www.netflixprize.com`.
43. DIETTERICH, Thomas G. Ensemble Methods in Machine Learning. In: *Multiple Classifier Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–15. ISBN 978-3-540-45014-6.
44. CUTLER, Adele; CUTLER, D. Richard; STEVENS, John R. Random Forests. In: *Ensemble Machine Learning: Methods and Applications*. Boston, MA: Springer US, 2012, pp. 157–175. ISBN 978-1-4419-9326-7.
45. DIETTERICH, Thomas G. An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization. *Machine Learning*. 2000, vol. 40, no. 2, pp. 139–157. ISSN 1573-0565.
46. ZHOU, Zhi-Hua. *Ensemble Methods: Foundations and Algorithms*. 1st. Chapman & Hall/CRC, 2012. ISBN 978-1439830031.
47. TITERICZ, Gilberto. *1st PLACE SOLUTION - Gilberto Titericz & Stanislav Semenov* [online]. 2015 [visited on 2018-06-26]. Available from: `https://www.kaggle.com/c/otto-group-product-classification-challenge/discussion/14335`.
48. ROSENBLATT, F. The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. *Psychological Review*. 1958, pp. 65–386.
49. SCHMIDHUBER, Jürgen. Deep learning in neural networks: An overview. *Neural Networks*. 2015, vol. 61, pp. 85–117. ISSN 0893-6080.
50. MEDSKER, LR; JAIN, LC. Recurrent neural networks. *Design and Applications*. 2001, vol. 5. ISBN 9781420049176.
51. DORFFNER, Georg. Neural Networks for Time Series Processing. *Neural Network World*. 1996, vol. 6, pp. 447–468.
52. BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*. 1994, vol. 5, no. 2, pp. 157–166. ISSN 1045-9227.
53. HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long short-term memory. *Neural computation*. 1997, vol. 9, no. 8, pp. 1735–1780.
54. GRAVES, A.; MOHAMED, A. r.; HINTON, G. Speech recognition with deep recurrent neural networks. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, pp. 6645–6649. ISSN 1520-6149.
55. YAN, Shi. *Understanding LSTM and its diagrams* [online]. 2016 [visited on 2018-06-27]. Available from: `https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714`.

56. CHO, Kyunghyun; MERRIENBOER, Bart van; GÜLÇEHRE, Çaglar; BAHDANAU, Dzmitry. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR*. 2014. Available also from: `http://arxiv.org/abs/1406.1078`.

57. CHUNG, Junyoung; GULCEHRE, Caglar; CHO, KyungHyun; BENGIO, Yoshua. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*. 2014. Available also from: `http://arxiv.org/abs/1412.3555`.

58. PASCANU, Razvan; GÜLÇEHRE, Çaglar; CHO, Kyunghyun; BENGIO, Yoshua. How to Construct Deep Recurrent Neural Networks. *CoRR*. 2013. Available also from: `http://arxiv.org/abs/1312.6026`.

59. BROWNLEE, Jason. *How to Make Baseline Predictions for Time Series Forecasting with Python* [online]. 2016 [visited on 2018-06-06]. Available from: `https://machinelearningmastery.com/persistence-time-series-forecasting-with-python/`.

60. GUO, X.; YIN, Y.; DONG, C.; YANG, G.; ZHOU, G. On the Class Imbalance Problem. In: *2008 Fourth International Conference on Natural Computation*. 2008, vol. 4, pp. 192–201. ISSN 2157-9555.

61. LONGADGE, Rushi; DONGRE, Snehalata. Class Imbalance Problem in Data Mining Review. *International Journal of Computer Science and Network*. 2013, vol. 2. ISSN 2277-5420.

62. CHEN, Weixuan; SAMUELSON, Frank W. The average receiver operating characteristic curve in multireader multicase imaging studies. *The British journal of radiology*. 2014, vol. 87 1040, pp. 20140016.

63. CHOLLET, Francois et al. *Keras* [`https://keras.io`]. 2015.

64. ABADI, Martin et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Available also from: `https://www.tensorflow.org/`.

65. AL IQBAL, Ridwan. Empirical learning aided by weak domain knowledge in the form of feature importance. *CoRR*. 2010. Available also from: `http://arxiv.org/abs/1005.5556`.

66. PEDREGOSA, F. et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, pp. 2825–2830.

67. SRIVASTAVA, Nitish; HINTON, Geoffrey; KRIZHEVSKY, Alex; SUTSKEVER, Ilya; SALAKHUTDINOV, Ruslan. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. 2014, vol. 15, pp. 1929–1958.

68. KINGMA, Diederik P.; BA, Jimmy. Adam: A Method for Stochastic Optimization. *CoRR*. 2014. Available also from: `http://arxiv.org/abs/1412.6980`.

69. BAYER, J. et al. On Fast Dropout and its Applicability to Recurrent Networks. *ArXiv e-prints*. 2013.
70. ZAREMBA, Wojciech; SUTSKEVER, Ilya; VINYALS, Oriol. Recurrent Neural Network Regularization. *CoRR*. 2014. Available also from: `http://arxiv.org/abs/1409.2329`.
71. NAGPAL, Anuja. *L1 and L2 Regularization Methods* [online]. 2017 [visited on 2018-06-12]. Available from: `https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c`.
72. PUMPERLA, Max. *Hyperas* [`https://github.com/maxpumperla/hyperas`]. 2016.
73. BERGSTRA, James. *Hyperopt: Distributed Asynchronous Hyperparameter Optimization* [`https://github.com/jaberg/hyperopt`]. 2013.
74. BERGSTRA, James S.; BARDENET, Rémi; BENGIO, Yoshua; KÉGL, Balázs. Algorithms for Hyper-Parameter Optimization. In: *Advances in Neural Information Processing Systems 24*. Curran Associates, Inc., 2011, pp. 2546–2554.

Appendix A

Acronyms

AI Artificial intelligence

ANN Artificial neural network

AUC Area under curve

GRU Gated recurrent unit

LSTM Long short-term memory

ML Machine learning

MLP Multilayer perceptron

MSE Mean square error

NN Neural network

RMSE Root mean square error

RNN Recurrent neural network

ROC Receiver operating characteristic

SVD Singular-value decomposition

Appendix B

Visualizations of predictive behavior

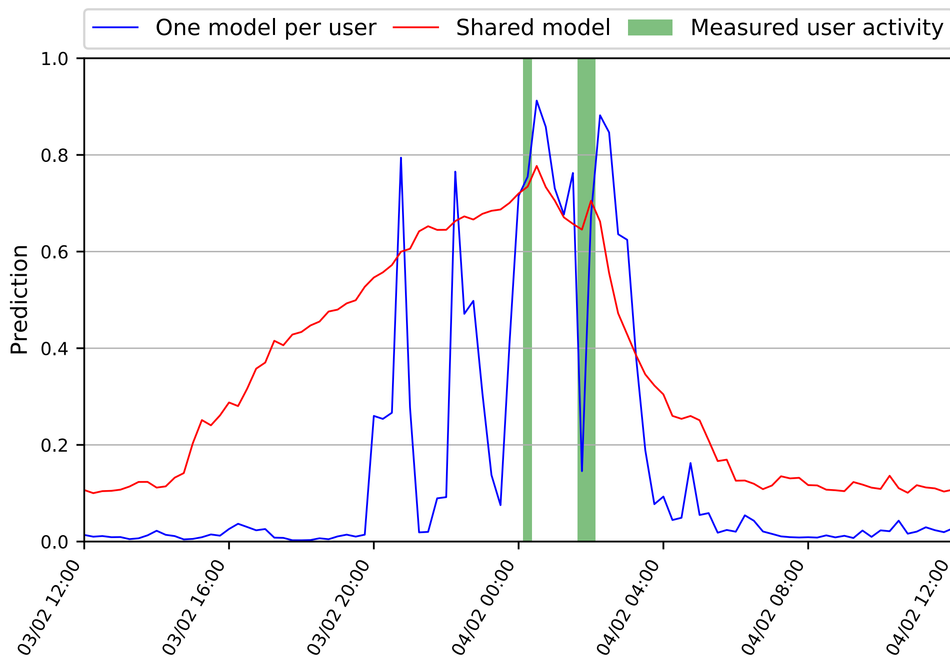

Figure B.1: Comparison of the behavior of LSTM model using different experimental settings

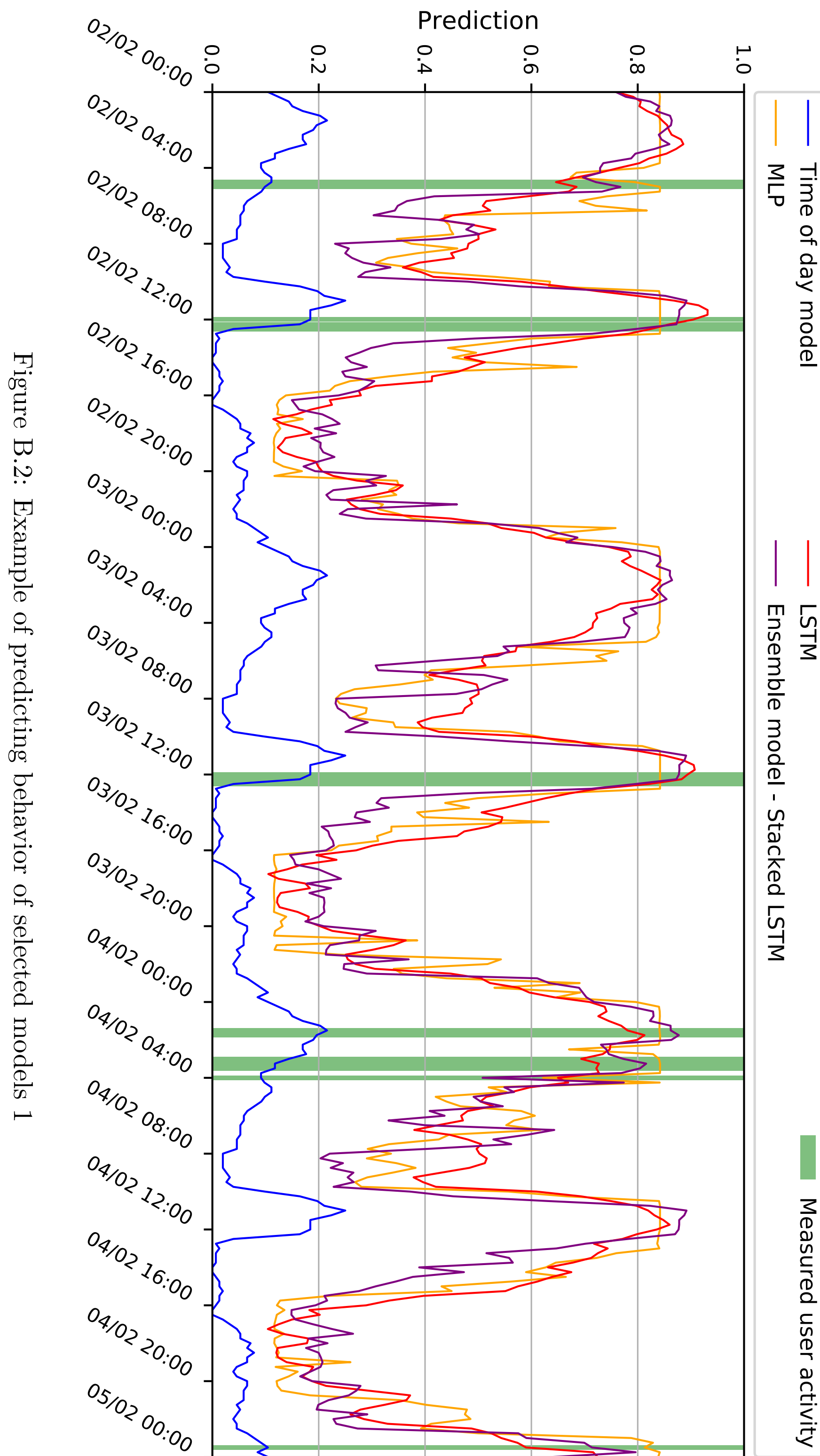

Figure B.2: Example of predicting behavior of selected models 1

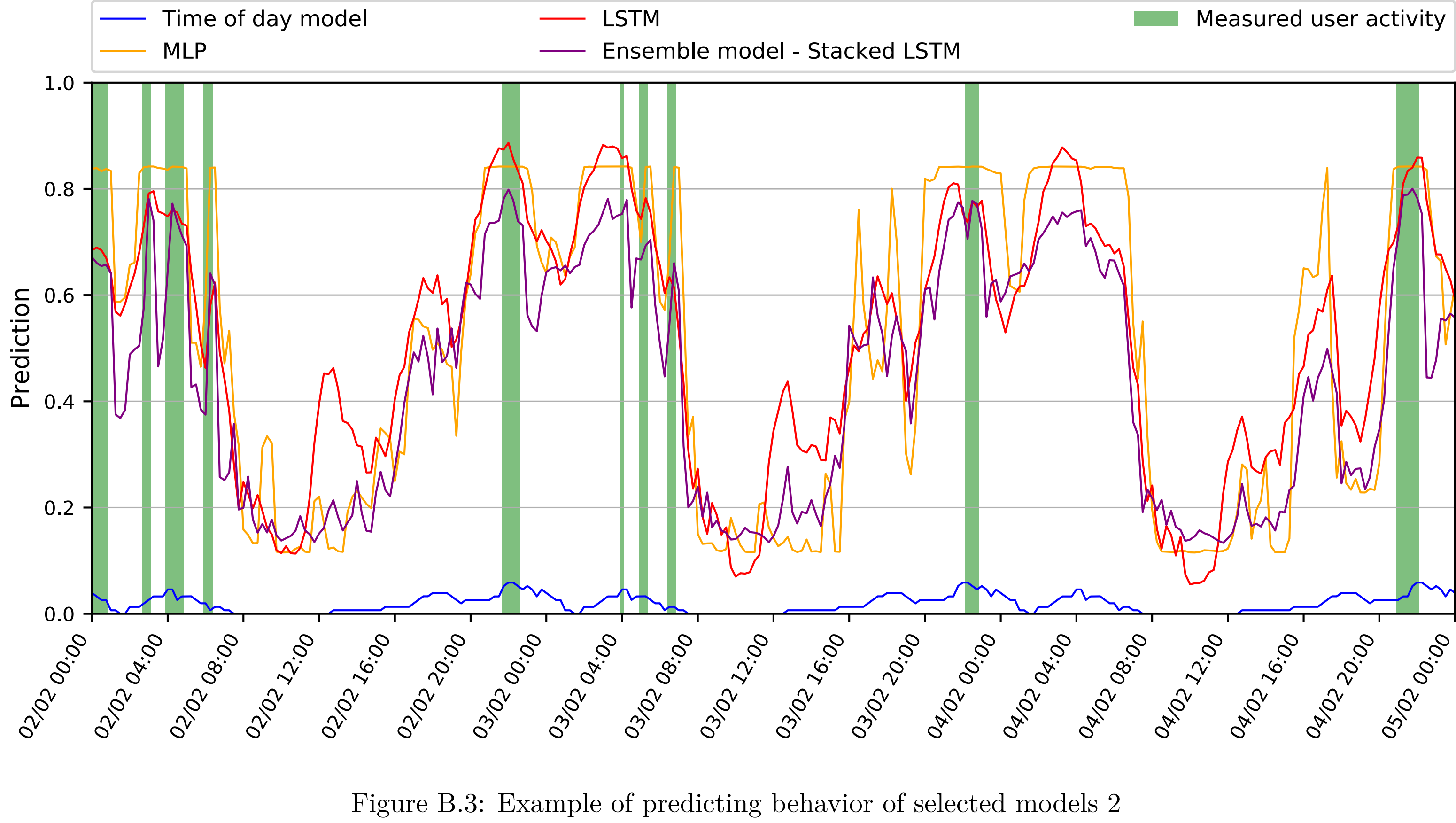

Figure B.3: Example of predicting behavior of selected models 2

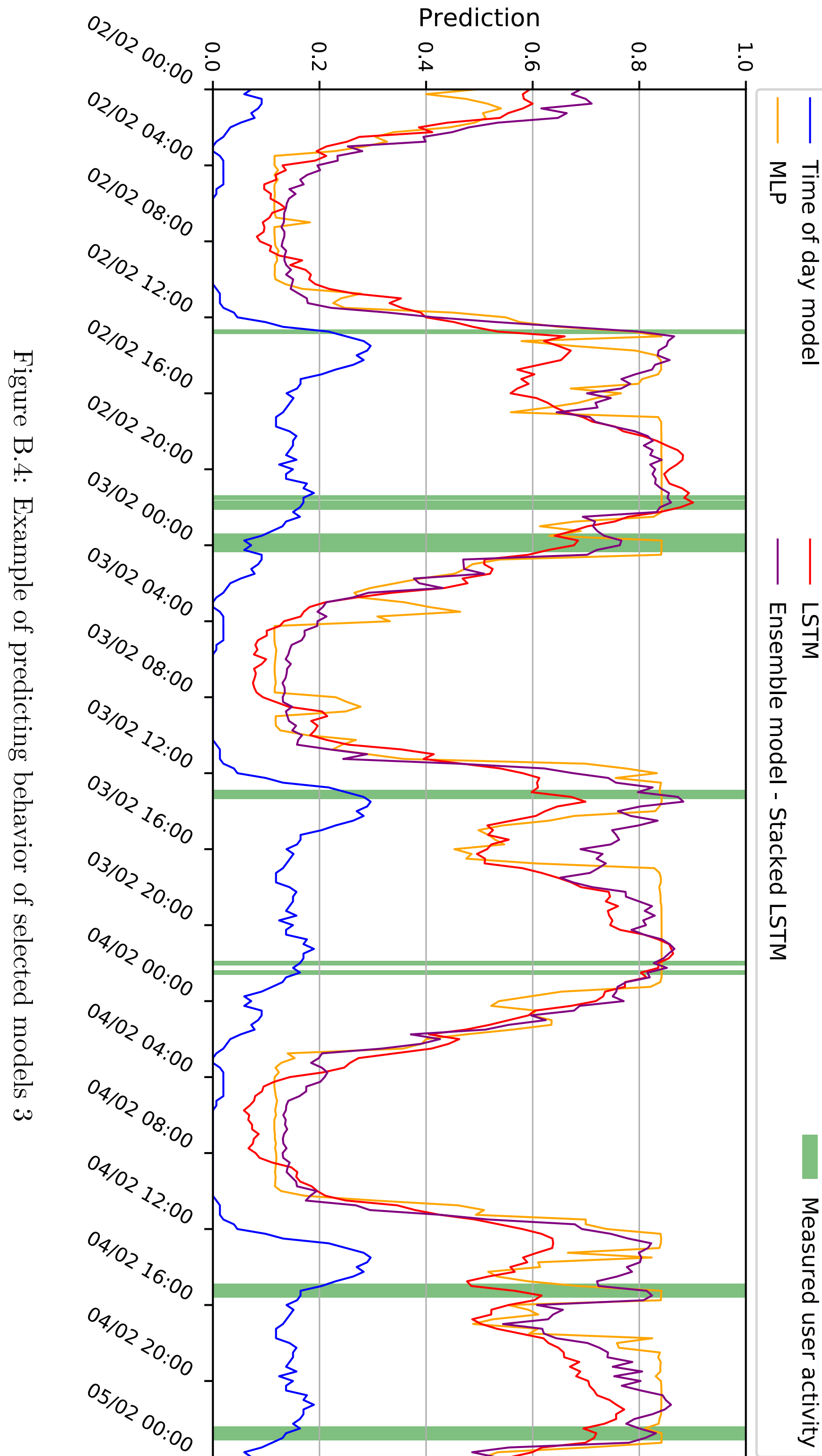

Figure B.4: Example of predicting behavior of selected models 3

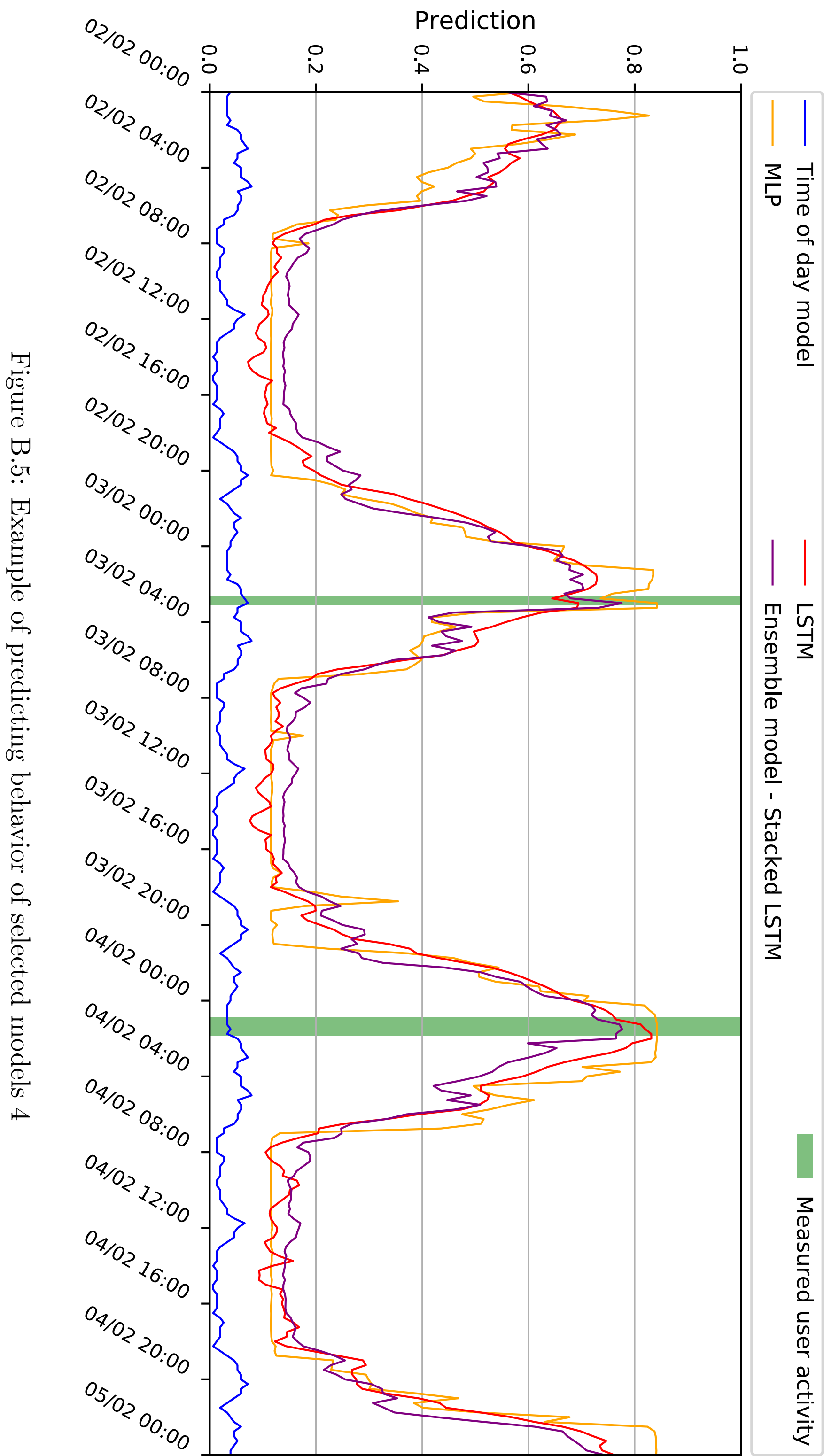

Figure B.5: Example of predicting behavior of selected models 4

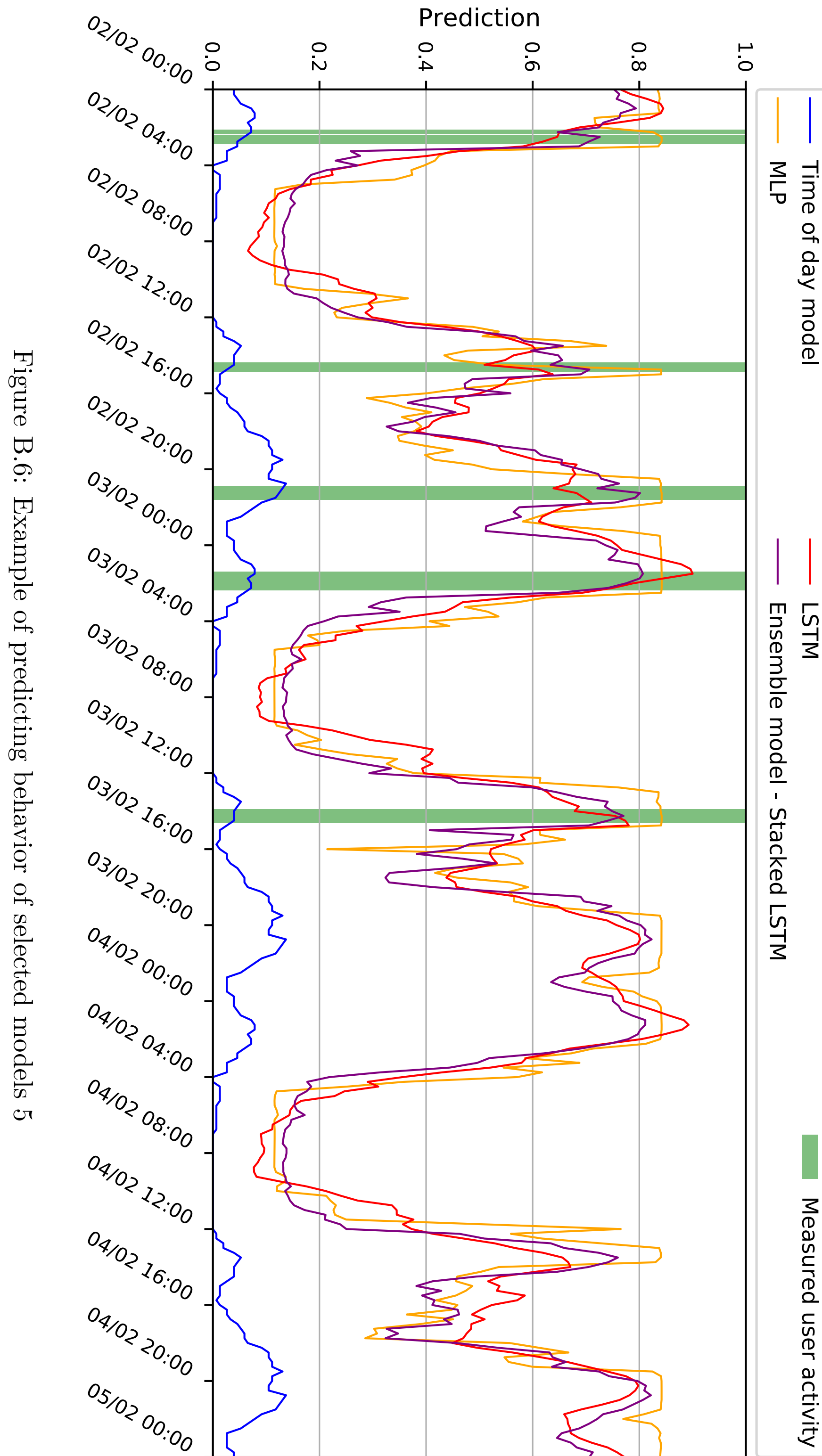

Figure B.6: Example of predicting behavior of selected models 5

Appendix C

Contents of attached storage medium

```
readme.txt ........................ file with contents description
thesis.pdf ........................ thesis text in PDF format
thesis_text ....................... directory of LaTeX source codes of the thesis
visualization ..................... directory containing a visualization application
```