FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Content-Based Recommendation Model Trained Using Interaction Similarity
Student:	Petr Kasalický
Supervisor:	Ing. Tomáš Řehořek
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2018/19

Instructions

Explore the area of Recommender Systems with focus on two major approaches: Collaborative Filtering and Content-based recommendation.

1) Investigate different methods of generating embeddings for item attributes of different data types (such as sets, textual descriptions, numbers, etc.)
2) Design and implement an algorithm capable of predicting interaction similarity based on embeddings of item attributes, using, for example, artificial neural networks. Implement item-based k-Nearest Neighbor recommendation model which uses the predictive model as the item similarity measure.
3) Evaluate the proposed model on multiple different datasets.
4) Discuss the contribution of interaction similarity prediction to standard Content-based recommendation models.

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 31, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Content-Based Recommendation Model Trained Using Interaction Similarity

Petr Kasalický

Department of Applied Mathematics
Supervisor: Ing. Tomáš Řehořek

May 14, 2018

Acknowledgements

First of all, I would like to thank my supervisor Ing. Tomáš Řehořek for all valuable information, time, patience and granted resources, including datasets and hardware used in my experiments. I thank participant of the Recombee Deep Learning Meetups, namely doc. Ing. Pavel Kordík, Ph.D., Bc. Radek Bartyzal, Ondřej Bíža and Ing. Ivan Povalyaev for a lot of advice and suggestions about machine learning. Finally, I thank my family and friends whom I have neglected because of this thesis.

Declaration

In Prague on May 14, 2018 ....

Citation of this thesis

Kasalický, Petr. *Content-Based Recommendation Model Trained Using Interaction Similarity.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstract

This bachelor's thesis describes the recommendation system and two major approaches, Collaborative filtering and Content-based recommendation. The new hybrid approach, which combines these two methods, is proposed. This method increases recall of content-based recommendation by up to 216% and allows more precise recommendation for newly added items, which suffers from the cold-start problem. This designed and implemented approach uses machine learning methods such as embedding or artificial neural networks, which will also be briefly introduced along with a way of evaluating the quality of the recommendation.

Keywords recommendation system, embedding, deep learning, artifical neural network, Python

Abstrakt

Tato bakalářská práce se zabývá doporučovacími systémy a jejich základními přístupy: Kolaborativní filtrování a Atributové doporučování. Je představen nový hybridní přístup, který kombinuje tyto dva přístupy. Tato metoda zvyšuje recall atributového doporučování až o 216% a umožňuje přesnější doporučování pro nově přidané věci, které trpí cold-start problémem. Tento navržený a implementovaný přístup využívá metod strojového učení jako je embedding nebo umělé neuronové sítě, které budou taktéž stručně představeny, spolu se způsobem vyhodnocování kvality doporučování.

Klíčová slova doporučovací systém, embedding, hluboké učení, umělé neuronové sítě, Python

Contents

List of Figures

List of Tables

Introduction

Nowadays, articles, videos, e-shop items, or songs and movies offered by streaming services are being added every day on the Internet. No one can go through this vast amount of available content, so recommendation systems become more important than ever before, as they help to pick only those relevant items for a particular customer. These systems, however, have some problems they have to deal with. One such problem is a cold-start problem, which in some circumstances prevents newly added items from being recommended.

This work presents a new hybrid method that solves this problem and thus increases the success of the recommendation systems. To fully understand this method, I first introduce the recommendation systems, their fundamental principles, usage, evaluation, and problems. Next, I will say what is embedding, present examples and highlight their advantages and disadvantages. Then I will briefly introduce the neural networks from the basics to the Deep Feed Forward Networks that are used in the proposed method. After clarifying this theory, I will design this new method with emphasis on data preprocessing, implement it in Python using technologies such as Jupyter, Keras, and PySpark, and in the final chapter I will publish the results on actual datasets of two e-shops and evaluate the success.

Goal

The aim of the research part of this bachelor thesis is to explain the importance of the recommendation systems and describe their two main approaches to recommending: Collaborative Filtering and Content-Based Recommendation. After familiarizing with the basic principles, I analyze the problems of these approaches with emphasis on the cold-start problem. Next, embedding is defined, and various embeddings for different data types (such as a text description, set, or numbers) are explored. After this introduction to RS and machine learning, the practical part of the thesis is to design and implement algorithm capable of predicting the interaction similarity of items through neural networks from created embeddings. This model will be used in the Nearest Neighbor algorithm for the recommendation, and evaluated in the light of the success of the recommendation on multiple different datasets that will be presented in detail. The results will be compared with traditional recommendations, and its contribution will be discussed.

Chapter 1

Analysis

1.1 Recommendation system

In this chapter, I will introduce what the recommendation systems are, why they are so important today and where is possible to meet them. I will also describe the principles of the functioning of the recommendation systems, introduce basic approaches such as collaborative filtering and content-based recommendation, describe the cold-start problem and finally explain how the quality of the recommendation algorithm can be evaluated.

A recommendation system, also known as recommender system, is a platform that tries to predict user's preferences for an item and allows to find relevant content for him. *"Recommendations made by such systems can help users navigate through large information spaces of product descriptions, news articles or other items."* [1]

These systems are widely used virtually wherever there is more content available. A typical example of service using the recommendation system is an e-shop that aggressively and continually endeavors to impose some merchandise through first screens, banners, emails, or other channels. Some form of the recommendation system can be found of course in giants such as Facebook that uses it, among other things, when selecting a relevant feed, or Google to suggests similar videos on Youtube. They will also find use in online newspapers, streaming services like Spotify, or even at finance. Also, *"a number of successful startup companies like Firey, Net Perceptions, and LikeMinds have formed to provide recommending technology."* [2]

There are two basic approaches to selecting from the vast amount of available content the one that is most interesting for a particular user. However, it is possible to combine these methods into so-called ensembles, which number is growing in practice due to better results. The primary goal of this work is to create a new hybrid approach.

1.1.1 Collaborative filtering

Collaborative filtering (CF) is the first of these basic approaches. It is widely used because of its versatility across different domains, as well as through its efficiency, accuracy, and scalability. This method uses the fact that user's behavior is not random, but there are some patterns in it. The primary concern when looking for content for a particular user is to find the user's most similar user and to inspire with his interactions. Interactions are thought to be some actions of the user in the system such as product view, rating, purchase, search, like or dislike, a recommendation to another user, add to cart or favorites, etc. Some value can be assigned to these actions indicating their importance, such as the purchase of the item is far more important than its mere view. All these interactions together define user. When RS looks for recommendations, it can find users who have the similar past and predict the future of one user according to the past of the other. Unfortunately, *"a collaborative filtering system must be initialized with a large amount of data because a system with a small base of ratings is unlikely to be very useful."* [2]

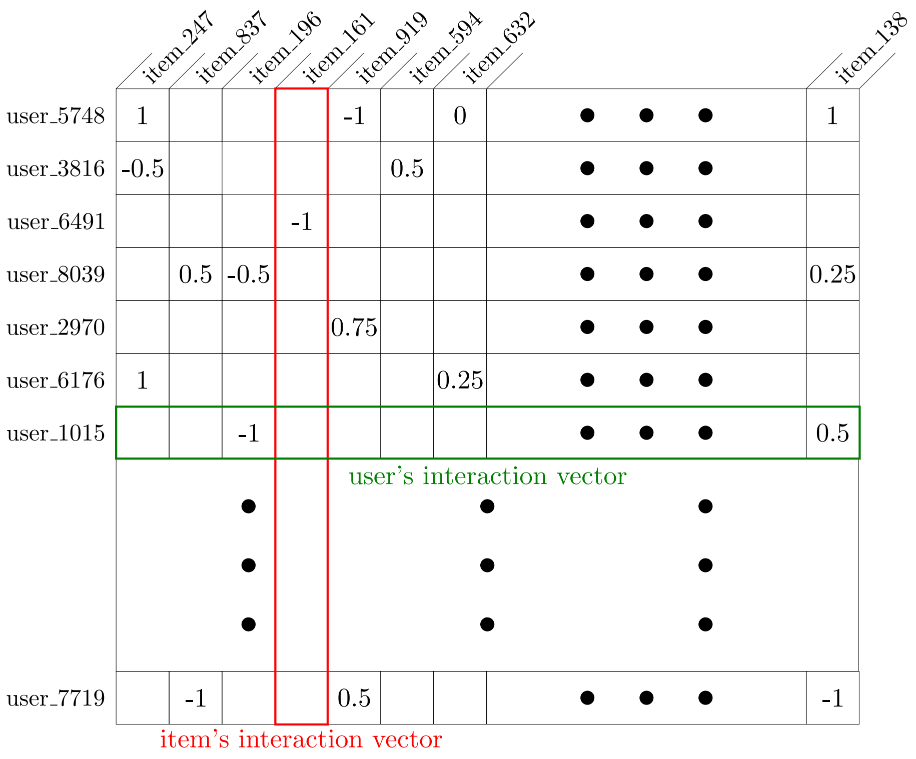
Now I will introduce the concept of the *user's interaction vector* and show how to get it. For simplicity, I only suppose interactions of the type of product view. I'll take a list of all the items on the platform, for example, all the articles in the newspaper or the products in the e-shop, and for each of them, I will put the number one in the resulting vector if the user has seen the item, otherwise, it is zero. I get a vector of size n, where n equals the number of all items. When interaction vectors are stacked, interaction matrices arise. I assume that all items are unique. Formally:

- U is sequence of all users
- $m = |U|$, number of all users
- I is sequence of all items
- $n = |I|$, number of all items
- M^i is set of items that user U_i has seen
- v^i is n-tuple for user U_i, also called *user's interaction vector*, where

$$\forall k \in \{1..n\} : v_k^i = \begin{cases} 1 & \text{if } I_k \in M^i \\ 0 & \text{otherwise} \end{cases}$$

- $V^{m \times n}$ is *interaction matrix*, where $\forall p \in \{1..m\} : V_{*,n} = v^p$

By definition, this matrix contains the *user's interaction vector* in each row, but if the columns are taken as vectors, the vector will be created for each item as well. I will call it the *item's interaction vector* and use it in my approach. The interaction matrix is also sometimes called the rating matrix and is usually huge but very sparse. This definition is limited to values zero and one, but in practice, the interaction matrix can contain any numbers, especially if RS takes into account other types of interactions than simple views. The rating matrix is not the only possible interpretation of the list of interactions, but it is undoubtedly the most used one. For example, unlike the time series recommendation, the information, when the interaction was performed, is not used. Example of real interaction matrix with more types of values and the appropriate vectors can be found in Figure 1.1.


user's interaction vector
item's interaction vector

	item_247	item_837	item_196	item_161	item_919	item_594	item_632		item_138
user_5748	1				-1		0	● ● ●	1
user_3816	-0.5					0.5		● ● ●	
user_6491				-1				● ● ●	
user_8039		0.5	-0.5					● ● ●	0.25
user_2970					0.75			● ● ●	
user_6176	1						0.25	● ● ●	
user_1015			-1					● ● ●	0.5
			● ● ●				● ● ●	● ● ●	
user_7719		-1			0.5			● ● ●	-1

Figure 1.1: Interaction matrix

Each user, therefore, has his sparse interaction vector that characterizes him. If RS looks for similar users, just needs to find similar vectors. The methods for measuring vector similarities used in collaborative filtering according to [3] are:

- Cosine similarity (COS)
- Pearson correlation coefficient (PCC)

"PCC calculates similarity as the covariance of two users' preferences (ratings) divided by their standard deviations based on co-related items." [3]

However, I will measure similarity of vectors using cosine similarity, which returns values from the interval $[-1, 1]$ and is expressed as the cosine of the angle between the two vectors. The formula is:

$$COS(v^i, v^j) = \frac{\sum_{k=1}^{n} v_k^i \cdot v_k^j}{\sqrt{\sum_{k=1}^{n}(v_k^i)^2} \cdot \sqrt{\sum_{k=1}^{n}(v_k^j)^2}},$$

This metric is widely used not only in recommendation systems but throughout machine learning. I will use it in my new approach, but I will not apply it to the *user's interaction vector*, but to the *item's interaction vector*, thus gaining similarity between items.

Of the stated formulas, you may notice the main benefits of collaborative filtering, that is the domain independence. There is no need for more profound information about users or items. It is enough for each of them to have a unique identifier. In this case, collaborative filtering differs from the approach that I will refer next.

1.1.2 Content-based recommendation

The second main approach is called the Content-based (CB) recommendation. This method requires knowledge of the recommended products. Not just identifiers like the previous approach, but an additional information is needed. Such information may be, for example, textual description of items, name, images, tags, category or binary content of the item, if it is the music, etc. First, I will explain the general principle of such a recommendation and then show how to handle a variety of additional information about items automatically, without the need of manual intervention.

Compared to collaborative filtering, where RS recommends what similar users liked, here RS is looking for items similar to those I liked. Suddenly, there is no need for a metric as similar users, but how are items similar. The first and easiest option I have mentioned is to use the *item's interaction vector* and cosine similarity. However, this approach has many potential problems, such as cold-start, which I will explain later. There are better and more accurate ways to capture the similarity of items. One approach, which is very demanding, expensive and inexplicable in practice, is to manually define relations between items. An e-shop administrator writes that dog and dog food are related to each other, and if the customer purchases a dog, the system should recommend a dog food. The recommendation will then only depend on how the administrator describes the relationship between products, which makes it very likely to miss unexpected coercion. That was is just for illustration. Of course that such systems are not used today. With the boom of machine learning, a whole range of automatic methods was developed to find similarities between all kinds of items.

As part of my project, I restrict myself to the idea that I have for each item a vector in space that best describes it. How to obtain such a vector is described in Section 1.2. So I can measure the similarity of vectors representing items as I measured the similarity of users.

1.1.3 Hybrid methods, cold-start problem

Besides such strictly separate methods, there are mixed ones that take something from both to generate better results. In general, a combination of several different models into one big better is called an ensemble, in the case of recommendation systems we talk about hybrid approaches, which are mainly designed concerning their problems. In cases where one system fails, another one will be used. [4]

A typical problem with CF is according to [5] a cold-start problem. This problem mainly concerns new items that have little or no interaction. According to [4], the cold-start problem is one of the biggest problems with which to deal with the recommendation systems. If the e-shop only recommends using CF and cosine similarity, new products without a single interaction will never be recommended. Content-based recommendation system, on the other hand, does not suffer from this problem because it does not use the interactions at all. Several cold-start solutions use machine learning methods such as a matrix factorization or deep learning and neural networks. Their list, including the description, can be found in [4]. This list will be complemented by a explanation of the Meta-Prod2Vec method, introduced in 2016 in [5]. However, for the description of Meta-Prod2Vec, it is necessary to explain some other principles, so it will be fully introduced in Subsection 1.2.5.

At the very end, I will introduce one more category, Knowledge-based recommendation systems. These programs are expert systems and require a specific interaction from the user, for example, displaying the decision tree and letting him click through, or requesting list of requirements from the user and then recommend. An example might be when a user wants to buy a house, he will fill out a form on the real estate website, and the system will suggest the house with the highest match of parameters.

1.1.4 Evaluation

I have already described several different methods of recommendation, but how to determine which one is more accurate and gives better recommendations? There are a couple of methods of evaluation, and none of them is standardized and ubiquitous. Nevertheless, I will show and describe one of the most used methods for evaluating the success and use it in my experiments. But first, I say the general division of the evaluation.

Evaluation can take place online or offline. In general, there is a much more conclusive online metric, where the success of the engine is tested on real users. An example of this can be Facebook, that has hundreds of versions all over the world. Generally, a huge traffic is needed, because people are split into the groups and different recommendation model is given to each group. Then the one with higher click-through rate is chosen as better.

In this work, I will use offline evaluation, that is, the evaluation using already collected data without the need for new ones. There are also methods somewhere between based on offline evaluation using artificially created users whose behavior is learned from the real ones using Reinforcement Learning. *Recall* and *catalog coverage* (CC) were chosen as offline metrics for this thesis.

Recall, also called *sensitivity*, is a general metric in the information retrieval calculated as the ratio of recommended relevant items to all relevant. According to [6], *recall* does not punish wrong recommendations, so if RS recommends all items, *recall* will be 100%. There is also a metric called *precision* (*confidence*) addressing this imperfection, which indicates how much data labeled as relevant was truly relevant. As a *catalog coverage*, the amount of recommended content will be measured. For example, RS can recommend bestsellers and nothing more, most of the customers will not mint, but the CC will be very low. Also, since RS's goal is to help the customer to discover new products, my effort will be to maximize the *recall* and CC in my experiments.

Calculation of *recall* as defined above is very trivial when it comes to classification. How to measure recall for recommendation? I will describe it in details. On input of the algorithm is required a model that measures similarity of two items. A random group of users, where each of them has interacted with more than one product and whose interactions have not been used in model learning, is also required. The *recall* for the model is then as follows:

For each user, a list of products interacted by him is taken. Each entry in this list can be considered as relevant to that user. Now one entry is hidden. For other products in the list, the distances to all products are calculated and multiplied by the user's rating for the given product. Those similarities are summed together and trimmed to k most similar. If there is a hidden entry in the gained list, one is written as result, otherwise zero. This step is executed for each item, the results are summed up and divided by the number of all items interacted by the user. Obtained value is the *recall* for particular user. The procedure is repeated for all users in the selected group and the average *recall* is returned. While counting the *recall*, CC can be calculated too. Just save all recommendation for each hidden item, join them to set, take the amount of this set and divide it by the total number of products to get CC. Formal description of this can be found in Algorithm 1.

1.2 Embedding

It is a well-known fact that the computer can handle numbers without any problems, but other data representations are incomprehensible to it. At the first sight, a human can distinguish the objects in the image, recognize covers

```
Data: set of all items ($I$),
      set of tested users ($U$),
      relation of interactions $r : U \times I \to \mathbb{R}$
Input: number of recommended items $k$,
       model ($M$), represented by relation of similarity $m : I \times I \to \mathbb{R}$
Output: recall and catalog coverage of model
$m^*(i,j) = \begin{cases} 0 & \text{if } i \neq j \\ m(i,j) & \text{otherwise} \end{cases}$
$A := \emptyset$ (set of recommended items)
$R := 0$ (recall)
$G := 0$
foreach $u \in U$ do
    $T := 0$
    $C := 0$
    foreach $h \in I : r(u,h) \neq 0$ do
        $f(i) = \sum_{j \in I : j \neq h} (m^*(j,i) \times r(u,j))$
        $S := (f(i_1), f(i_2), \dots, f(i_n))$
        $L :=$ indexes of $k$ highest values in sequence $S$
        if $h \in L$ then
            $T := T + 1$
        end
        $C := C + 1$
        $A := A \cup L$
    end
    $R := \frac{R \times G + \frac{T}{C}}{G+1}$
    $G := G + 1$
end
$CC := \frac{|A|}{|I|}$ (catalog coverage)
Return $R$ (recall) and $CC$ (catalog coverage)
```

Algorithm 1: Measurement of *recall* and *catalog coverage*

of one song, or find the same information in different grammatical interpretations. The computer cannot do this by itself. Some computer science disciplines try to teach a computer to perceive things as a human. One of the most significant breakthroughs in last couple years is Computer vision, which attempts to learn computers to see as people using advanced image processing. [7] Other is Nature Language Processing (NLP), which allows to build voice assistants, translators, etc.

Each of these disciplines, including recommendation systems, must transform their objects of interest, such as images or videos, into vectors of real numbers, because most of the machine learning methods are designed to work with vectors. According to [8], this transformation is called embedding. High-quality embedding should also reveal the similarity between real objects and is able to transfer it to n-dimensional space. Embeddings are the absolute foundation for creating a high-quality recommendation system. [9]

Ways of embeddings are many and continually growing. Probably for every type of object (image, text) there already exist some embedding. In this section, I am going to show the embeddings of text, numbers, and sets, but first I will introduce how to visualize the embedding result and thereby evaluate its quality. At the end I describe the Meta-Prod2Vec method promised in Subsection 1.1.3.

1.2.1 t-SNE

To maintain information about objects, most embeddings return a high-dimensional vector. It's not a problem for a computer, and all machine learning works in a high-dimensional space, but a human cannot imagine it and verify that similar objects are truly mapped to neighboring areas.

Fortunately, the T-distributed stochastic neighbor embedding (t-SNE) algorithm was introduced in 2008. This method is *"capable of retaining the local structure of the data while also revealing some important global structures (such as clusters at multiple scales)."* [10] In practice, all that is needed to be provided are the high-dimensional vectors, the target dimension (usually 2 or 3) and a pair of hyperparameter. Unfortunately, t-SNE is very sensitive to hyperparameter setting. How to appropriately choose hyperparameters and get the desired result is greatly described in [11]. This algorithm will be used to compare the embedding qualities with respect to interactions. t-SNE belongs to the dimensionality reduction techniques in addition to PCA or matrix factorization.

1.2.2 Words

Because of the use of text as a general media, it is no wonder that word embeddings are among the oldest and most discussed. According to [12], the first attempts to manually translate text into vectors took place in the 1950s, automatic feature selection techniques then came in the 1980s. Of a large number of such methods, I have chosen three:

- Bag-of-words (BoW)
- Hashing Vectorizer (HV)
- Paragraph Vector (doc2vec)

While describing the following algorithms, I assume that I have a document (list of sentences) for each input item and the output is a vector of real numbers. The number of items equals n.

Bag-of-words is the oldest of these methods. The first use of this term is noted in the 1954 in [13]. However, it is only an expression, the algorithm itself was introduced later. This technique has, with minor modifications, a general use when processing the discrete objects to vectors. The algorithm proceeds first by going through all the sentences and splitting them into words. These words can be lemmatized (converted to basic form) but it is not necessary. The first step of the algorithm is to create a dictionary containing all used words. Next, each document is taken and converted to the vector with size equal to the number of words in the dictionary. For each word in the dictionary represented by the position in the vector, the frequency of the word in the sentence is written. As a result, the vectors have for each position a number signaling the count of represented word occurrences in each sentence. These vectors, like the dictionary, are usually very large (e.g., 100,000) and very sparse (contain 99% zeros). In addition to lemmatization, it is possible to make other adjustments to the text such as correcting misspellings, converting to lowercase, or removing stop words (and, with, or, etc.). This method does not reflect the order of the words in the sentence, the synonyms and other linguistically significant phenomenas. *“For example, ”powerful”, ”strong” and ”Paris” are equally distant.”* [14] Two steps of BoW, without lemmatization or any other modification, are illustrated in Figure 1.2.

Typically, a term frequency–inverse document frequency (tf-idf) transformation is applied to BoW embedding, which determines how the individual elements of the vector (words) are relevant for the document. It works by reducing the weight of words that occur in most documents (such as stop words) and increase it to unique words. Implementation differs slightly across applications, but the basic procedure is as follows. *“Given a document collection D, a word w, and an individual document $d \in D$, we calculate*

$$w_d = f_{w,d} \cdot \log \frac{|D|}{f_{w,D}}$$

where $f_{w,d}$ equals the number of times w appears in d, $|D|$ is the size of the corpus, and $f_{w,D}$ equals the number of documents in which w appears in D.” [15]

The tremendous size of the dictionary and the resulting vectors may negatively affect the memory and algorithm speed requirements. A way of compress this dictionary and vectors called LSA will be shown at the end of this section. The compromise is the Hashing Vectorizer, which is capable of generating vectors of the desired length n. It works by hashing words to one of the number $[0, n)$. There is no need to create an extensive dictionary, just hash each word and enter the number of occurrences at the appropriate vector position. It can happen that a position contains the sum of multiple words, especially for a small n. [16] Great advantage over BoW is the ability to process new documents containing unique words without having to recalculate all other documents.

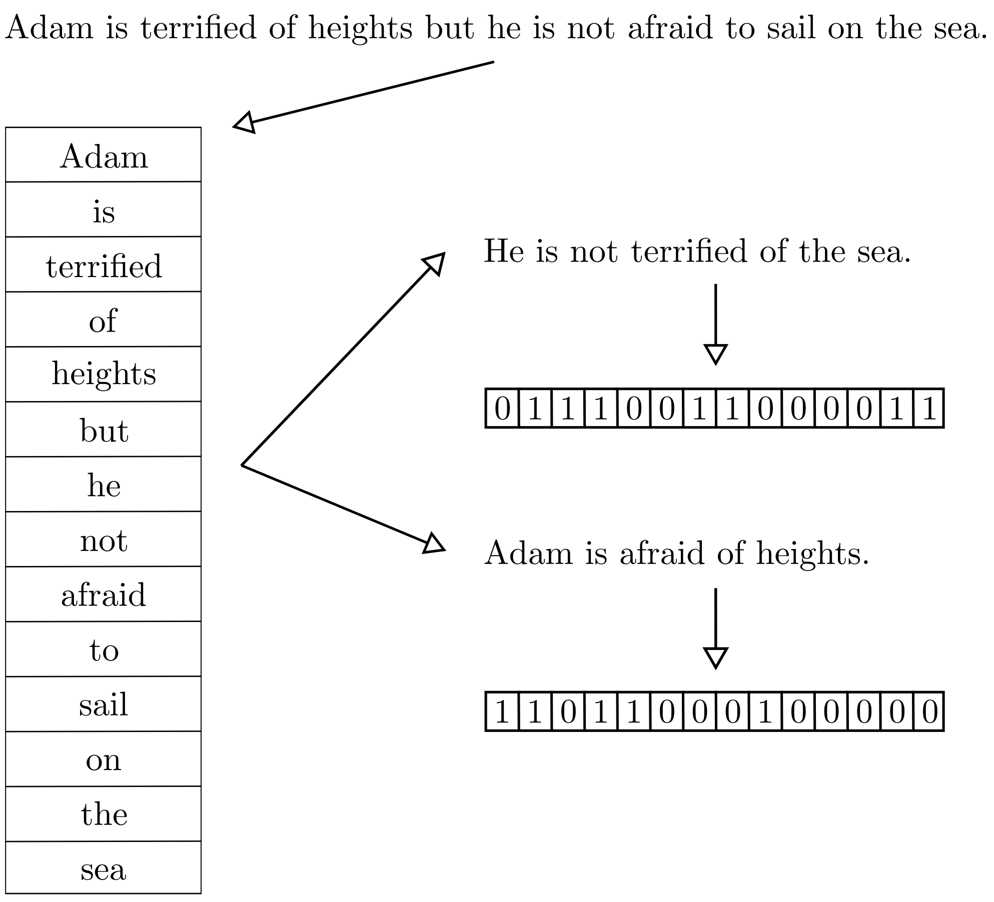

Adam is terrified of heights but he is not afraid to sail on the sea.
Adam
is
terrified
of
heights
but
he
not
afraid
to
sail
on
the
sea
He is not terrified of the sea.
0 1 1 1 0 0 1 1 0 0 0 0 1 1
Adam is afraid of heights.
1 1 0 1 1 0 0 0 1 0 0 0 0 0

Figure 1.2: Illustration of bag of words

Finally, I describe the doc2vec method, which was introduced under name Paragraph Vector in 2014 in the article *Distributed Representations of Sentences and Documents*. [14] Tomas Mikolov builds on his work and the word2vec method which he introduced a year before in [17]. Therefore, to understand doc2vec, it is necessary to first explain word2vec.

As I have already mentioned, bag-of-words suffers from the loss of semantics. All words are equally distant from each other, although it is not in natural language. Word2vec allows for each word to find its numeric representation while capturing relationships such as synonyms or analogies. [18] During this process it uses two algorithms that work the opposite to each other. The first one is called Continuous bag-of-words (CBoW), and it differs from the standard BoW in that it takes the neighborhood where the word is found (context). Explicitly, the Feedforward Neural Net Language Model (NNLM) takes this context as input and tries to predict that word. Thanks to this step, meaning (and representation of words) depends on the order in the sentence. The second model is the Skip-gram, which works very much

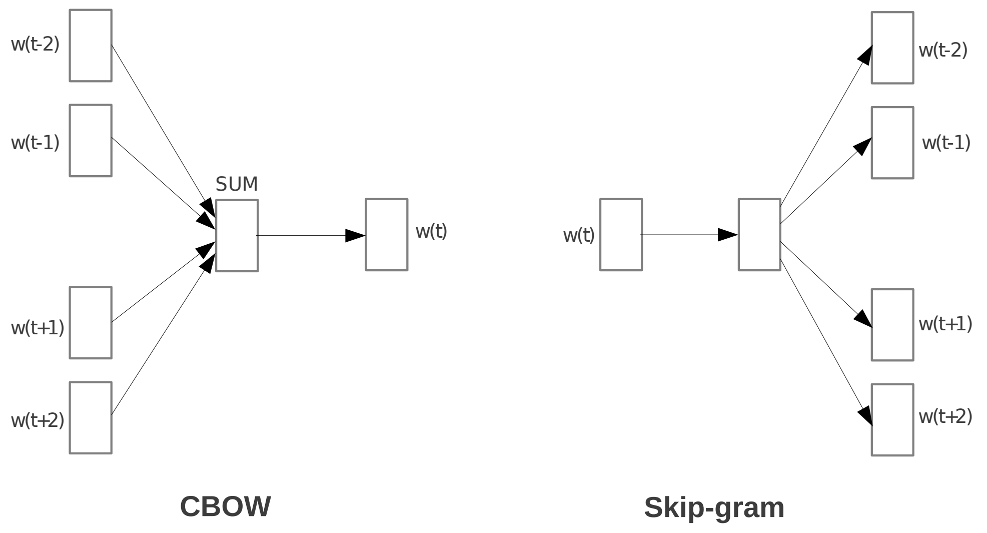


Figure 1.3: Two steps of word2vec [17]

like CBoW, just does not return the word according to the context, but the context according to the word. Both algorithms are illustrated in Figure 1.3. The greater the amount of text is given to word2vec, the more accurate it is. The hyperparameters of this method are the context size (the number of words around), the target vector dimension, or the length of the training. There are freely available models trained on data from Wikipedia or Google News. [17], [18], [19]

This model is able to return a vector representation for each word. The linearity of these words also applies, i.e., $queen + man = king$. But how to build embedding of the whole text? Before doc2vec was introduced, it was common practice to take the vectors of each word and join them into one vector using some operation (sum, average). Now when word2vec has been described, the explanation of doc2vec is trivial because its learning uses very similar algorithms. The CBoW model, which had on the input the context of the word to predict, now also processes the input vector referred to as paragraph-id. The value of this vector does not truly matter. It is just the identifier for the paragraph (or any other part of the text). Such a model is called the Distributed Memory version of the Paragraph Vector (PV-DM). The Skip-gram model is modified so that there is no input word and the output context but takes only paragraph-id, and the content is modeled. Both steps are illustrated in Figure 1.3. [18]

1.2.3 Sets

Set embeddings are far more straightforward than word embeddings, as well as a variety of written literature about both topics. In recommendation systems

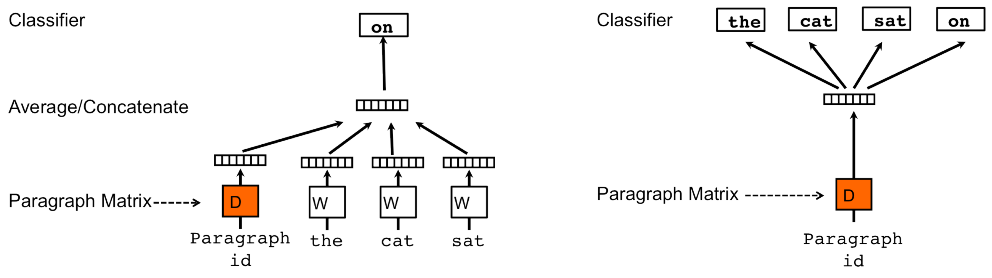


Figure 1.4: Distributed Memory (left) and Distributed Bag of Words (right) versions of the Paragraph Vector [14]

are typically stored in sets different categorization of items, tags or even names. All fields that do not make much sense to ask for their own meaning, but they are more about labeling. This fact is also used for embedding. The word embedding should reflect the meaning of words and map semantically nearby words to close vectors. Of course, the sets are depending on the content, but most of the uses mentioned above does not have a separate meaning, and the elements are semantically equally distant. A model that did not reflect semantics and only took into account the presence of content has already been introduced. Speech is about bag-of-words. It is a little confusing, but it is possible to use BoW, even though the content does not have to be words at all. An example may be a set of identifiers for a category where the whole bag is a list of all identifiers used. In the word embedding, the resulting vector contains the number of occurrences of a word in a piece of text, but in the case of sets, is captured only the presence (1) or the absence (0) of the element in the set. A huge and sparse vector might arise again.

1.2.4 Numbers

As I mentioned earlier, embedding is needed because most machine learning methods assume vectors of real numbers to input. A number can be considered as a vector in 1D space, especially after standardization. However, I will show two basic embeddings of numbers. Both consist of dividing the numerical axis into bins and then assigning numbers to these intervals. This method is called as discretization or binning. Interval sizes can be the same, then we talk about equal-width, or they can contain approximately the same number of items (equal-frequency). A very sensitive parameter, how many bins to produce, is required. The number of bins equals the size of the resulting vector. It is not possible to say that one method is better for every single case but in most cases, it is recommended to use an equal-frequency method that works better with outliers. But it always depends on the nature of the data. Both ways have their advantages and disadvantages. For example, for data where there is an uneven number of nominal values (ratings 1, 2, 3, 4, 5), there is no reasonable equal-frequency distribution. [20]

Here ends the list of embeddings for basic data types. Boolean processing does not need to be commented. In addition to basic data types, it is possible to create embedding for whole items as well. The suggested approach includes one, but I will introduce another one called Meta-Prod2Vec.

1.2.5 Meta-Prod2Vec

Meta-Prod2Vec has already been mentioned in Subsection 1.1.3. It is embedding, which takes into account product attributes as well as interactions. It builds on and expands the Prod2Vec method proposed in [21] a year earlier. The reason I put it down to the end of this chapter is its association with the word2vec method, specifically with its Skip-gram algorithm. Prod2Vec proceeds interactions including their timestamp. It is possible to sort the products as they were viewed by a particular user. This sequence gives a "sentence" for each user. The list of sentences is proceeded by the Skip-gram model, which returns the vector for each "word" (product). From the description, it must be clear that Prod2Vec also suffers from a cold-start problem because it dependents on interactions only. Therefore, this method has been extended to Meta-Prod2Vec, which, in addition to interactions, also takes into account product metadata (attributes). *"Because of the shared embedding space, the training algorithm used for Prod2Vec remains unchanged. The only difference is that, in the new version of the generation step of training pairs, the original pairs of items are supplemented with additional pairs that involve metadata."* [5], [21]

1.2.6 Latent semantic analysis

Latent semantic analysis (LSA), method introduced in 1988, improves information retrieval by reducing dimensionality. It focuses on revealing the relationship between the used terms, especially in bag-of-words, such as synonymy, homonymy, or polysemy. "[22] *showed that people generate the same keyword to describe well-known objects only 20 percent of the time.*" LSA tries to find these different expressions describing one object and merge them. Input is a term-document matrix (build by bag-of-words), which contains raw term frequencies in its cells. On this matrix is applied a tf-idf or similar operation to get the characteristic expressions for the documents. The most important step is a dimensional reduction by matrix factorization, specifically singular value decomposition (SVD), that is able to decompose the matrix into a multiplicity of three others. The middle of these three matrices contains expressions *"sorted in decreasing order"*. Next, a truncated SVD algorithm is applied, which means that it takes only k highest values and their corresponding vectors. As a result, each expression can be represented by a vector of the k dimension. [23]

1.3 Artificial neural network

"Although the first articles about Artificial Neural Networks (ANN) were published more than 50 years ago, this subject began to be deeply researched on the early 90s, and still have an enormous research potential." Everyone has probably heard of them lately, as their signature can be found under most new methods of artificial intelligence. Also, they help solve the problems of other disciplines. Applications are found in biology, medicine, finance, transport, military, law, and many others. Their great advantage over classical models is the ability to find non-linear dependencies. One example I have already introduced is word2vec, which uses neural networks in both inner algorithms to predict word and context. ANNs must be variable to have so many applications. Each neural network consists of smaller elements. How these elements are stacked and what algorithms are used, defines network's properties and usage. You can see an overview of the architectures of the networks in Figure B.1. Simple Feed Forward Network is great for explaining basic principles. All the information in this chapter, including an introductory quotation, is from the book *Artificial Neural Network, A Practical Course.* [24]

1.3.1 Basics

Neural networks have been inspired from the very beginning by the structure of a human brain. The first paper describing the neural computational model was written in 1943 by McCulloch and Pitts. The result was the creation of the first artificial neuron. Like its biological template, this neuron had multiple inputs called dendrites $(x_1, \dots x_n)$, one output called the axon (y), and the body where the computation is performed. Body consists of the so-called activation function (g) applied to the activation potential (u), which equals to the weighted sum of inputs (with weights $w_1, \dots w_n$) adjusted for bias (θ). Formally:

$$y = g(\sum_{i=1}^{n} w_i x_i - \theta) = g(\sum_{i=0}^{n} w_i x_i) \ for \ x_0 = -1; \ w_0 = \theta$$

Inputs are invariant, just like activation functions, and learning of neurons is through weight and bias (also called threshold) changes. A more detailed description of learning will be given below. There is only one axon, but is able to branch out. That allows neurons to be connected to larger system that exist in the brain as well. Simply connect output (axon) of a neuron to the input (dendrite) of another neuron to create a neural network. There are many ways

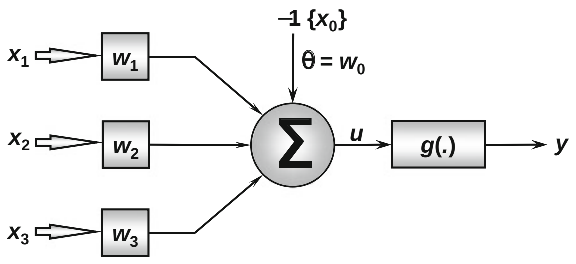


Figure 1.5: The artificial neuron [24, p. 12]

of connecting neurons. For Feed Forward architecture we talk about linking to the layers. Labeling of these layers varies, but I will distinguish these:

Input layer is not made of any neurons, but provides input for the next layer. Technically it is only a vector $(x_1, \dots x_n)$.

Hidden layer can be zero, one or hundred times in the ANN and allows more complex calculations.

Output layer is the last layer, which combines an output of neurons to provide output vector of the whole network.

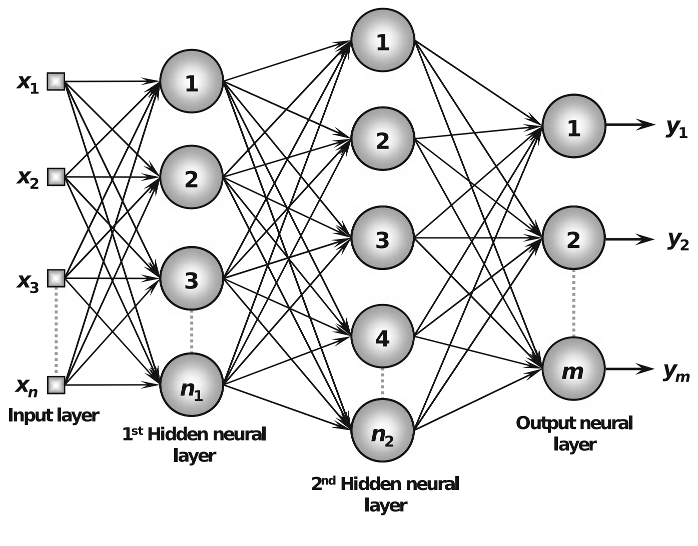


Figure 1.6: Example of a feedforward network with multiple layers [24, p. 23]

As can be seen in Figure 1.6, each neuron in the l_i layer is connected with its output to input of each neuron in the l_{i+1} layer. This way stacked layers are also sometimes referred to as fully connected layers. The number of such layers is just one of many hyperparameters in the Deep Feed Forward Network. The others will be introduced in the following sections.

Such interconnection is the main reason for the existence of the already mentioned activation function because its task is to normalize the output of the neuron. Activation functions add complexity to neural networks because without them, the multilayer network could be summed up to one layer. The activation function is required to be fully differentiable for the purpose of learning. There are justifiable cases where they are only partially differentiable, but I will not deal with them. Here are three examples of commonly used and fully differentiable activation functions:

Logistic function produces a real number in the range $[0, 1]$ and is expressed by the mathematical formula:

$$g(u) = \frac{1}{1 + e^{-\beta u}}$$

where β is a constant declaring the slope. Special case, when $\beta = 1$, is called the sigmoid function.

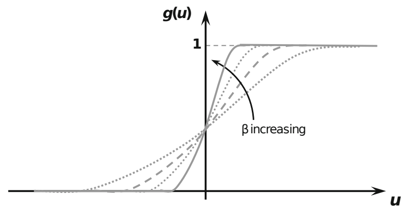


Figure 1.7: The logistic function [24, p. 16]

Hyperbolic tangent function is very similar to the logistic function but provides values in the range $[-1, 1]$. Its mathematical expression is:

$$g(u) = \frac{1 - e^{-\beta u}}{1 + e^{-\beta u}}$$

with the same meaning of β as above.

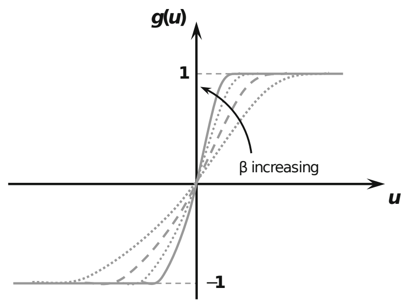


Figure 1.8: The hyperbolic tangent function [24, p. 17]

Linear function, also called identify function, is against reasons listed above, why to use the activation function, but in certain justified cases is used, usually when a full range of output on the last layer is wanted. For completeness, its formula is:

$$g(u) = u$$

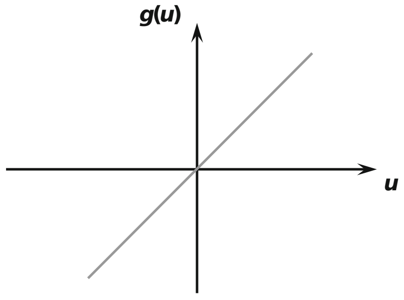


Figure 1.9: The linear function [24, p. 17]

This list contains only the basic functions. There are many more. Other example could be a Gaussian function or a group of ReLU functions, whose popularity has been rising for the benefit of faster convergence.

1.3.2 Training

One of the main advantages of ANN is their ability to learn. For learning Forward Networks is needed not only input, but also the desired output (supervised learning). The network tries to figure out what the relationship between input and output is. That allows *"generalizing solutions, meaning that the network can produce an output that is close to the expected output of any input values."* The training process consists of the following partial steps:

1. calculate the output $(y_1, \dots y_n)$ from the input for current setting of weights and bias
2. compare the obtained output with the desired one $(\hat{y}_1 \dots \hat{y}_n)$ through the loss function and get an error
3. propagate an error back to the network and change weights (including bias)

How to calculate network output from input has already been shown. I will only add that this phase is also called forward propagation. The difference between the calculated and desired output is indicated by another of the hyperparameters, namely the loss function. The choice of loss function depends on the nature of the problem. Some function is selected for the classification and another for the regression problem. I will introduce Mean squared error (MSE), which is used extensively for regression problems. Its mathematical expression is:

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Calculated error is used in the third step called backpropagation. This algorithm was introduced in 1974 by Paul Werbosen and caused a significant breakthrough in learning. It uses, among other things, the derivation of the activation functions to determine the effect of the weight W_{ji} on the output. This weight is adjusted for the next iteration $(t + 1)$ with the formula:

$$W_{ji}(t+1) = W_{ji}(t) + \eta \cdot q_{ji}$$

where η is the learning rate, that indicates the step size. The learning rate can be changed during the calculation, typically starting at a higher value when exploring the space, and gradually decreasing to find the global minimum. These changes can be controlled manually, but there are also so-called optimizers that change the learning rate automatically. Perhaps the most popular are the optimization algorithms Adam or SGD. The search of the value of weight W to get minimal error is shown in Figure 1.10.

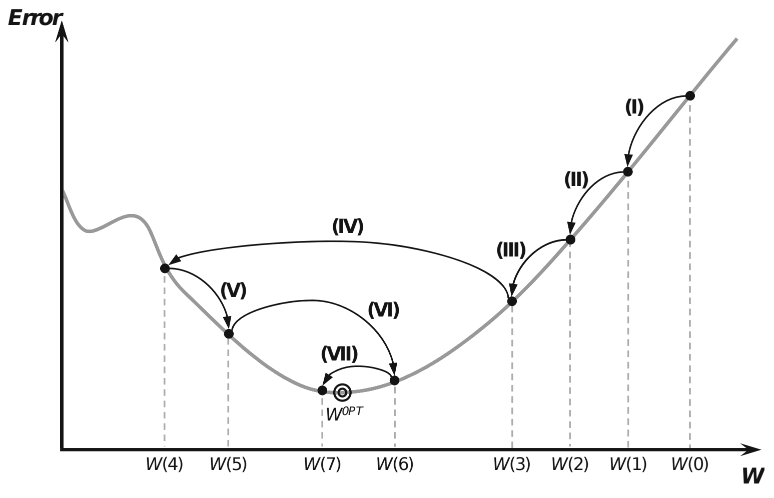


Figure 1.10: Changes of weight during learning [24, p. 74]

The variable g_{ji} reflects the weighting of W_{ji} on the error and the direction (sign) of the change. Its calculation includes partial derivatives of activation functions, varies according to whether it is an output or hidden layer and its full description is beyond the scope of this work. For shallow nets, this is a very accurate calculation, but for very deep nets, due to the massive number of variables, it is difficult to propagate the error from the output to the first layers. It is possible to use tricks such as residual connections, but I will not take care of them here.

Training of NN is an iterative process that includes these three steps over and over. The m input vectors $(x_1, \dots x_n)$ and the desired outputs $(\hat{y}_1, \dots \hat{y}_n)$ are required for learning. The dataset needs to be randomly divided into training and test (validation) data. The first one is used to train the network, the other to evaluate the ability of the network to generalize. Because forward and backward propagation can be implemented by matrix multiplication, it is possible to calculate outputs for multiple rows from dataset at once. This is used in learning because evaluating each element separately and adjusting scales would be terribly inefficient. For smaller datasets, it is possible to take the entire training dataset. For larger is used batch learning, when a fixed number of samples is taken (e.g., 512), passed through the network, the average error is calculated, and then the weights are adjusted. When all the training data is used, the epoch ends. Training is completed by the condition or after the execution of a defined number of epochs.

1.3.3 Testing

The aim of the training NN is not only to minimize the result of the loss function (error) calculated on the training data. From the network is wanted much more, namely to recognize patterns and rules between input and output. Deep neural networks are capable of incredibly complex calculations but are also very sensitive to overfitting. That is a situation where the network is not able to generalize. It does not find any patterns, but simply by setting hundreds of weights returns the desired output, but is unable to cope with new input. You can find the example of results of the correctly fitted network (a) and overfitted network (b) in Figure 1.11.

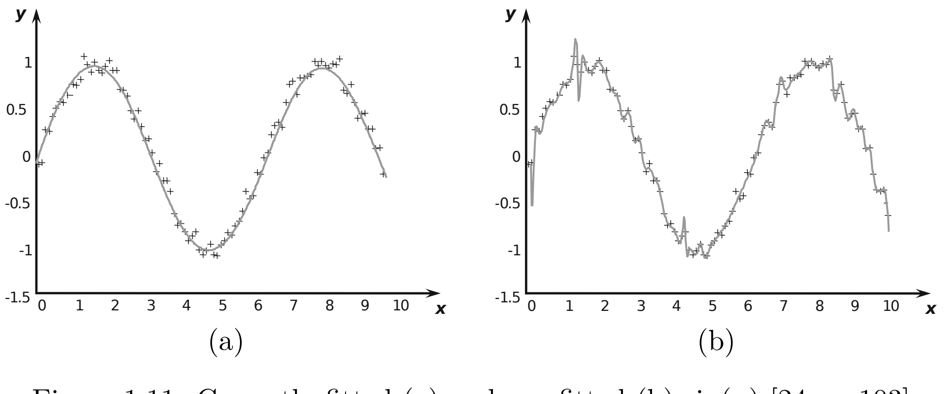


Figure 1.11: Correctly fitted (a) and overfitted (b) sin(x) [24, p. 103]

Evaluation the ability to generalize is provided by test subset of the dataset that the neural network must not use for learning. Test dataset is given to input of NN that calculates output and error but no longer propagates the

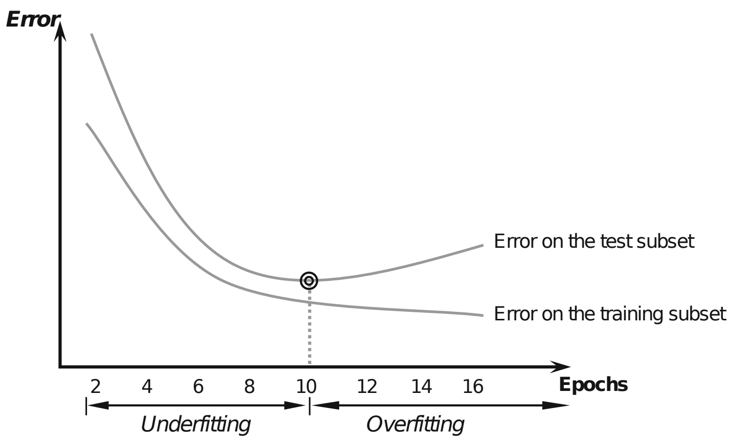


Figure 1.12: Underfitting and overfitting [24, p. 102]

error back, so the weights remain unchanged. Then, errors for training and testing subset are compared. The traditional course of these two errors during training is described in Figure 1.12.

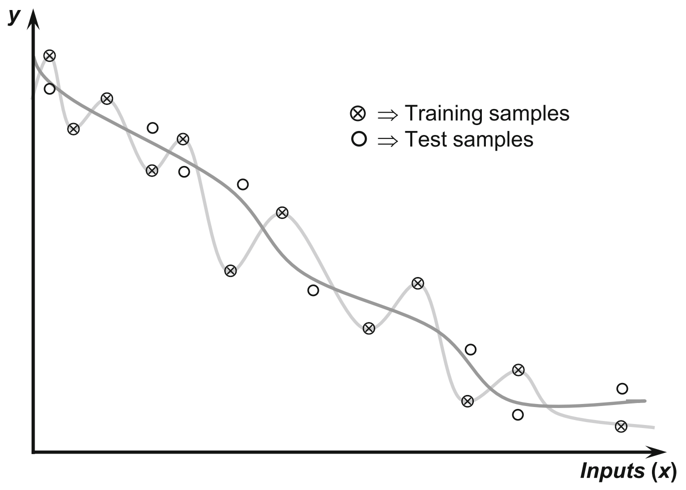


Figure 1.13: Example of overfiting [24, p. 104]

The moment, when the error on the test data starts to grow, and the network begins to overfit, can come in the tenth or even thousandth iteration. It depends on data and NN topology. Due to the vulnerability of NN for overfitting, a number of techniques have been developed to try to eliminate or at least to delay overfitting as much as possible. The list of the most popular methods is:

L1 and L2 regularizations increase the error by adding a sum of weights to returned loss and thus forces the weights to have low values. For MSE and the linear activation function on the last layer with the addition of

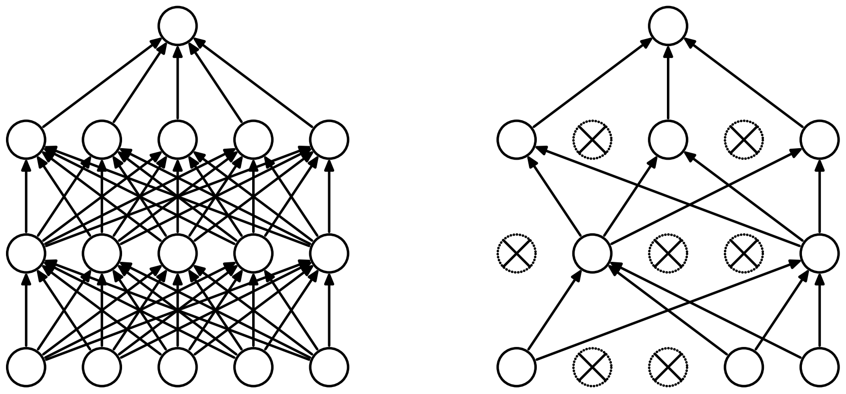


Figure 1.14: Neural net before and after applying dropout [25]

L2 regularization, the resulting error can be written as

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - \sum_{j=0}^{m} x_{ij}w_j)^2 + \lambda \sum_{j=0}^{m} w_j^2$$

where λ is another, very sensitive, hyperparameter. L1 regularization works the same way, only instead of the sum of the quadrates of weights uses the sum of the absolute values of weights. [26], [27]

Dropout method randomly skips neurons in hidden layers, including their connection, during the training phase. That *"prevents the units from co-adapting too much."* You can see the demonstration in Figure 1.14. Choosing which neurons to omit, can take place once for the whole epoch or better for each batch separately. The number of omitted neurons is given by the hyperparameter. Dropout is not used when evaluating test data. [25]

Batch normalization is primarily designed to accelerate the calculation but also has a regularization function. As input data is normalized, *"batch normalization normalizes the output of the previous activation layer by subtracting the batch mean and dividing the batch standard deviation."* It is recommended to use it in combination with a dropout. [28], [29]

1.3.4 Hyperparameters

I have already mentioned many hyperparameters, that is, the possibility of setting up a network that is invariant in the training process. In addition to the fact that training itself is an iterative process, the design of network is also iterative. There is no general procedure to determine the correct setting of the hyperparameters for a particular problem. There are only recommendations for specific situations. The hyperparameter list depends on the chosen architecture. For FFN, the following are the primary ones:

- Number of layers and neurons per layer
- Activation function
- Loss function
- Optimizer
- Regularization

The procedure for selecting hyperparameters along with the results will be listed in Section 3.4.

Chapter 2

Design

All the necessary theory is described, so I can now propose new hybrid recommendation method. First, I will explain its main idea and describe approach from a high-level perspective. The new approach is designed to address the cold-start problem described in Subsection 1.1.3 as a fundamental lack of the collaborative filtering. Technically it is an extension of a content-based recommendation where attribute information along with interactions contributes to determining similarity. The goal of this method is to teach the neural network to predict interaction similarity using the embedding of items. For a schematic of the method, see Figure 2.1.

To train the FNN, I need to build a dataset of inputs and outputs. The whole process is described in the next section. When the dataset is ready, it is necessary to design NN and iteratively choose hyperparameters. At the end of this chapter, I will use the output model of the trained network to recommend, and measure its quality by the already presented *recall*.

2.1 Data preprocessing

Data preprocessing is an essential part of this method, and therefore I will describe it in detail. The entry point of my work is dataset containing items, their attributes and interactions. The output of this section is a training set prepared for the input of a neural network.

There is a little problem with terminology here because until now the term dataset was meant to be the data prepared for the input of the neural network and their corresponding outputs. Now, this term has been extended to all data (products and their information and interactions) originating from one domain. Therefore, the data prepared for the network will be now referred to as a training dataset.

Name	Description	Categories	Price
Beachshorts	Elastic quick-dry bermudas ...	{'casual','men'}	4.89
Military Jacket	Army coat for men ...	{'cotton','khaki'}	38.98

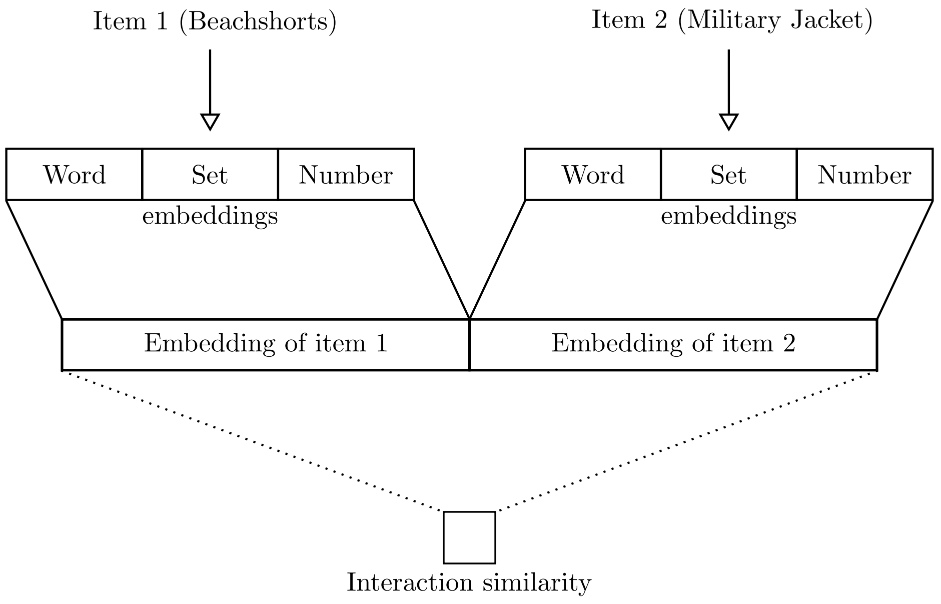


Figure 2.1: Illustration of proposed method

2.1.1 Embedding of each product

I begin by creating an embedding of each product in the dataset. I assume that the product information includes text (name, description), numerical data (price, number of pieces in stock) and sets (category, brand). For each of these attributes, embedding is created. These data types go through the following embeddings.

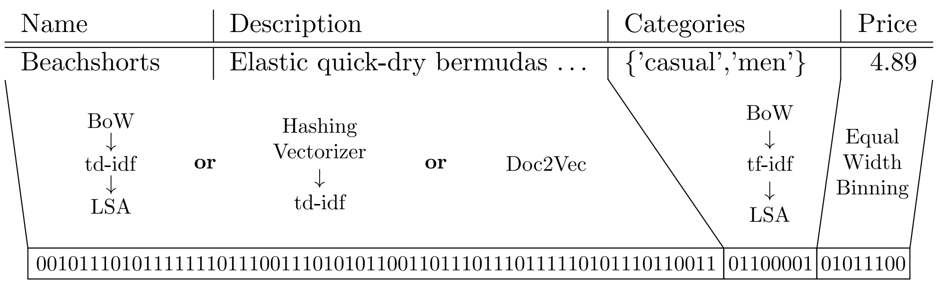
Because of sharing the dictionary between the individual text attributes, they are all joined, and one vector is retrieved for all of them together. In this work, I compare all three word embeddings listed in Subsection 1.2.2, namely Bag-of-words, Hashing Vectorizer and Doc2Vec.

BoW and HV are further regulated by tf-idf to reduce the stop words effect and highlight characteristic words. Since I require a vector of predefined size for the input of NN, the LSA method introduced in Section 1.2.6 is also applied in case of BoW. That allows all text attributes to be transformed into one vector of size n. The experiments are performed for $n = 64$.

Numeric attributes are not joined together like text but are processed individually by the equal-width binning method. Again, there is an option to set the size of the resulting vector, that equals the number of bins at discretization. Here I have chosen 8 to be the width of each numerical attribute.

The sets go through exactly the same transformation as the words, that is BoW → tf-idf → LSA. The only difference is that they do not build a common dictionary for all set attributes, but each attribute has a separate one. As with numbers, the resulting vector for each set attribute has a width of 8.

Now embeddings are ready for each attribute, and it is time to get embedding of the whole product. To preserve all information, the summing or averaging of the vectors is not chosen, but they are simply concatenated. The resulting embedding will then have a width of $64 + 8i + 8j$, where i equals the number of numeric attributes and j equals the number of set attributes. You can find an example of such concatenation in Figure 2.2, where vectors are limited to binary values for clarity, but in reality contain real numbers.



Name	Description	Categories	Price
Beachshorts	Elastic quick-dry bermudas ...	{'casual','men'}	4.89

Figure 2.2: Concatenation of embedding of each type

2.1.2 Interaction similarity

Dataset contains a list of interactions. The types of observed interactions and their weights are:

- Detail view, 0.25
- Purchase, 0.75
- Cart addition, 0.75
- Bookmarks, 0.75
- Rating

For each type of interaction there is a list of triplets $(user, item, weight)$, where $weight$ equals the explicitly given weight. The rating does not have weight because it contains the value, which the user has rated the product. These lists of triplets for each type can be combined into one large list. Since there is required only one value for each $(user, item)$ pair, $weight$ in this list is summed up for each unique pair $(user, item)$. The maximum result is set to 1, so $weight = min(1, weight)$. This is illustrated in Figure 2.3. From this list of interactions, a very sparse interaction matrix is constructed according to the definition and algorithm listed in Subsection 1.1.1. The matrix contains the *item's interaction vector* for each product with at least one interaction. Products without any interaction are not present.

Detail views

user	item	weight
235	486	0.25
288	883	0.25
275	83	0.25
…	…	…

Bookmarks

user	item	weight
288	597	0.75
270	83	0.75
218	486	0.75
…	…	…

Cart additions

user	item	weight
275	966	0.75
288	83	0.75
218	444	0.75
…	…	…

Purchases

user	item	weight
275	83	0.75
238	883	0.75
275	486	0.75
…	…	…

Ratings

user	item	weight
263	83	0.5
238	883	-1
288	597	1
…	…	…

$\longrightarrow$

Interactions

user	item	weight
275	83	1
238	883	-0.25
288	597	1
288	883	0.25
218	444	0.75
270	83	0.75
235	486	0.25
275	966	0.75
263	83	0.5
218	486	0.75
275	486	0.75
…	…	…

Figure 2.3: Joining more types of interactions

There are typically, besides users, crawlers, which visit all the products and index them, in this matrix. Their interactions interfere with the pattern of behavior of average users, and their effect is undesirable. I designate a crawler like a user who has interacted with more than $\frac{1}{4}$ of all products, and remove it from the matrix. Next, I put aside the users on whom the target model will be tested. Therefore, the rows (users) of this matrix are shuffled, and the part of the matrix is cut off and stored separately. I will refer to these users as *unused users*. The size of the cut-off is dependent on the total number of users and the desired precision of the measurement. I separate 5% of users. To see how the final model recommends for products that never saw, it is needed to shuffle and separate some of the products (*unused items*) as well.

2.1.3 Dataset

Interaction matrix along with embedding of all products is ready. I create the training dataset by taking all the products from the interaction matrix (except *unused items*) and tagging them as *used items*. From them, I create pairs with each other, even product with itself, connect their embeddings and compute the interaction similarity of them. The number of generated records is $|used\ items|^2$. You can see an illustration of this pairing in Figure 2.4, where

$Sim(x, y)$ is a cosine interaction similarity calculated from the remaining interaction matrix. Again, for clarity of the illustration, the vectors contain only zero and one. All these records build training dataset.

Item ID	Name	Embedding
8461	Beachshorts	0101101100
7812	Military Jacket	1101001011
9743	Biker Jeans	0111011010

Input	Output
01011011000101101100	Sim(8461, 8461) = 1.00
01011011001101001011	Sim(8461, 7812) = 0.53
01011011000111011010	Sim(8461, 9743) = 0.19
11010010110101101100	Sim(7812, 8461) = 0.53
11010010111101001011	Sim(7812, 7812) = 1.00
11010010110111011010	Sim(7812, 9743) = 0.38
01110110100101101100	Sim(9743, 8461) = 0.19
01110110101101001011	Sim(9743, 7812) = 0.38
01110110100111011010	Sim(9743, 9743) = 1.00

Figure 2.4: Building dataset

The output of the entire data preprocessing is a created training dataset and a list of users with interactions that were not used for measuring the interaction similarity (*unused users*).

2.2 Training

The data is almost ready. There is the last thing left before designing the neural network. In Subsection 1.3.3 I have described how to test NN functionality. It is necessary to put aside data that will not be used for training, but for testing the network and its generalization capabilities. As a last part of the data preparation, it is needed to randomly mix the entire training dataset and divide it into training and validation subset. Sometimes they are divided into a training, test and validation parts, where the latter is used to compare the models with each other, but this is not necessary because I will compare the models according to the achieved *recall*. The division ratio is dependent on the size of the dataset. The larger the validation subset, the more accurate the measurement, but the fewer data to train, and vice versa. I used 10% of the dataset as validation in my measurements.

Now is the time to design a NN. I use Deep Feed Forward Neural Network with 15 layers. The number of layers was set after few iterations. With more layers (> 20), the network had a learning problem, and with less (< 10) did not achieve such results. Other hyperparameters that I have found after few iterations and have not changed since in my experiments are:

As an activation function for all neurons in hidden layers, I have chosen the sigmoid. For the output layer containing only one neuron, it has been selected a linear activation function.

As a loss function has been chosen Mean squared error (MSE) because the problem is regressive by its character.

Optimizer = Adam

Batch size = 1,024

On the other hand, it took many iterations to find other hyperparameters like the method of regularization or the number of neurons on each hidden layer. The process of choice of these hyperparameters is described in Section 3.4. Batch normalization along with the 25% neuron dropout for each hidden layer is used in experiments. The number of neurons in hidden layers was established as the twice of the width of one training sample.

The network is assembled. Now I need to train it, test it on validation subset and evaluate its use in the recommendation.

2.3 Recommendation

The aim of this work is not only to train the network but also use it in the recommendation system. Quality of a model is measured by *recall*, described in Subsection 1.1.4. The number of recommended items (k) in this algorithm equals 5. A trained neural network is used as a model, that gives the similarity of two products. As a sample of users, for which is *recall* measured, a part of the interaction matrix named *unused users* is used. For measurements, users with more than one interaction are needed. I have randomly selected 250 users from this sample. Of course the more significant number of users is better, the measurement is more accurate, but it would also take much longer. It is not problem for a one-time evaluation, but I measure the *recall* regularly during the training to see how network learning affects the recommendation.

Training and evaluation took place in steps. I train the NN for 50 epochs (along with measuring validation error) and calculate *recall* along with *catalog coverage* of model. This step is repeated 20 times. At the end, the network was trained for 1,000 epochs. The measurement results along with the hyperparametrization course are in the next chapter.

CHAPTER 3

Experiments

In this chapter, I will first introduce two real datasets which were used for evaluation of the proposed method. Then I will demonstrate quality of chosen embedding using t-SNE. Finally, I will present the results of the proposed method.

3.1 Technology and hardware

Before I go to the measurement results, I have to briefly introduce the used technologies. Python is the most popular language in the academic world and machine learning specially, so it cannot be a surprise that I have also used it to implement proposed method. The main advantage of this language is the hundreds of available libraries and their interdependence. In the included source code, you can find the NumPy, PySpark, Pandas, Scikit-learn, SciPy libraries, or the Polyglot library for tokenization in word embeddings. I used the Keras API library for high-level work with neural networks implemented in Tensorflow. That is the software used. As far as hardware is concerned, the entire experiment was run on a computing server with the following configuration:

CPU Intel® Core^{TM} i7-6700

RAM 64GB

GPU GeForce® GTX 1080

What is important is the presence of a high-quality graphics card that computes all the neural network calculations (training, evaluation, recommendation). NN can also be trained on a processor that has a much smaller overhead to prepare data for the calculation, so it makes sense to use it to pass a single record, but on more extensive data, it is significantly slower. Conversely, counting the GPU records one by one would be incredibly inefficient, as it would take over most of the time to send and retrieve the data. That is why I introduced the batch, set its size to 1,024, and I train NN by proceeding this many records at once. The processor provides all other calculations including all preprocessing of data.

3.2 Data exploration

I have tested the proposed method on actual datasets of two e-shops. I will introduce both of them briefly. I add that if I talk now about interactions, then it is about the resulting triplets after merging the multiple types described by Figure 2.3.

Dataset A comes from a healthy nutrition store in the USA. It contains 2,725 products that were visited by 1,401,937 users and left 2,948,952 interactions. These values include crawlers. Products have the following attributes besides of their ID:

- Text: name, description, department, primary category, brand
- Number: rating (integer), price (real), sales rank (integer)
- Set: categories, collections

Some text attributes, such as brand, would be better to take as sets, but the supplied dataset looks like this, and the proposed embedding must be able to handle it.

Dataset B offers home furnishings in Brazil, so its language is Portuguese. This entire dataset is too large for available hardware, as it contains over 8,000 products. For the entire experiment, I chose 2,500 with the highest number of interactions. Together they have 3,253,753 interactions created by 2,502,613 users. The following attributes are available:

- Text: name
- Number: price (real), discount price (real), brand (integer)
- Set: categories

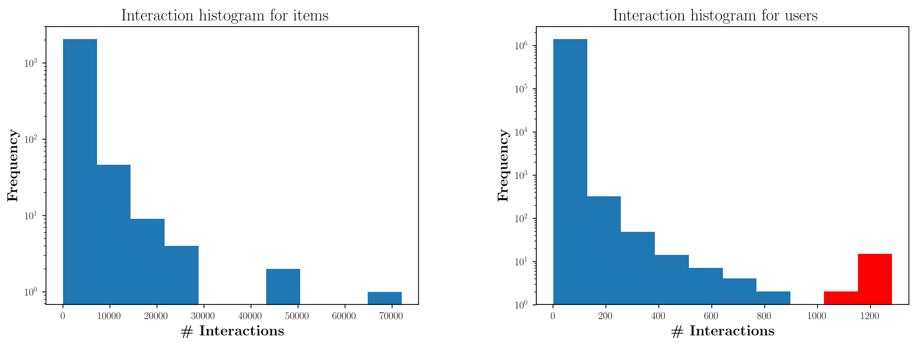
The brand is a numeric value because it is a reference to another table that I do not have. It does not matter because the number is similarly informative as the brand name. It would be better to perceive it again in sets. To summarize and compare these statistics, see Table 3.1.

The interaction count histogram is very similar for most of these datasets. A handful of users have more interactions, but most of them have one or two. The distribution of interactions in dataset A can be seen in Figure 3.1. In the user interaction histogram (b), you can also find crawlers whose interactions are separated from the interaction matrix. Similar outliers can be detected for products, but I do not remove them. It may be a bestseller or

Table 3.1: Datasets

	Dataset A	Dataset B
Products	2,725	2,500
Users	1,401,937	2,502,613
Interactions	2,948,592	3,253,753

some additional service offered by e-shops for every merchandise. Histograms for dataset B look almost identical thus are not shown.



(a) (b)

Figure 3.1: Interaction histograms for items (a) and users (b) for dataset A. Crawlers are marked with red.

3.3 Embeddings

In the experiment, one of my goals is to show that product attributes have some information about user interactions, and the involvement of multiple attributes leads to the more accurate recommendation. Therefore, training datasets, which take only a part of their attributes, are created and tested. Testing different types of embedding take place in the following steps:

1. A training dataset is formed only from text attributes as described in Section 2.1.1 and the best word embedding is selected.
2. Next training dataset contains the word embedding selected in the first step, and it is added set embedding of each product.
3. The last dataset extends the second one by containing numbers processing.

The quality of used embedding methods can also be demonstrated by projecting the resulting vectors into 2D using t-SNE. If embedding is proper,

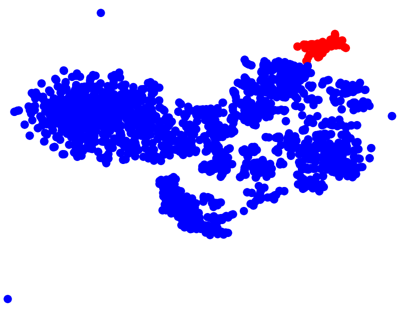


Figure 3.2: Clusters of items by interactions

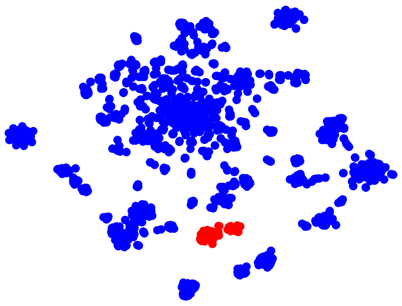


Figure 3.3: Clusters of items by embeddings

products should be in clusters. Similarly, it is possible to project *item's interaction vectors*, where clusters arise depending on how users have interacted with the products together. Since the dimensions of these vectors in a dense representation are equal to the number of users, which is a huge number, it is not possible to use only t-SNE, that is able to handle the 100-dimensional vectors playfully, but it is unusable for such large numbers. But I can use LSA to reduce the dimension to 100 and then apply t-SNE.

In Figure 3.2 you can see clusters of products from the dataset A according to the interactions, in Figure 3.3 the clusters according to the embedding of all the attributes. The marked red cluster contains the products belonging to the category called Essential Oils. It is obvious and no surprise that these products are associated together with interactions. Similarly, embedding seems to be of good quality because it has transformed the text into vectors and keep relationships between them. Details of both of these clusters can be found in Figures 3.5 and 3.4.

3.4 Hyperparameters

Most of the hyperparameters have already been set in Section 2.2. Problem was to determine the number of neurons on individual layers. I tried to assemble the pyramid with 128-64-32-16-8-4-2-1 or set the fixed layer size to 50–250 neurons. Since the length of the input vectors is variable depending on the processed attributes, I have chosen also variant number of neurons in the hidden layers, specifically twice the size of the input vector, which equals to four times the size of embedding. As a result, this means that if I process only the text, each hidden layer has 256 neurons. For example, embedding containing all attributes for dataset A has a dimension of 104, so a number of neurons equals 416. Output layer connected to last hidden layer has in this experiment always only one neuron with linear activation function.

The worst hyperparameter regarding finding was a regularization. You can see learning outcomes in the form of a graph of the evolution of training and

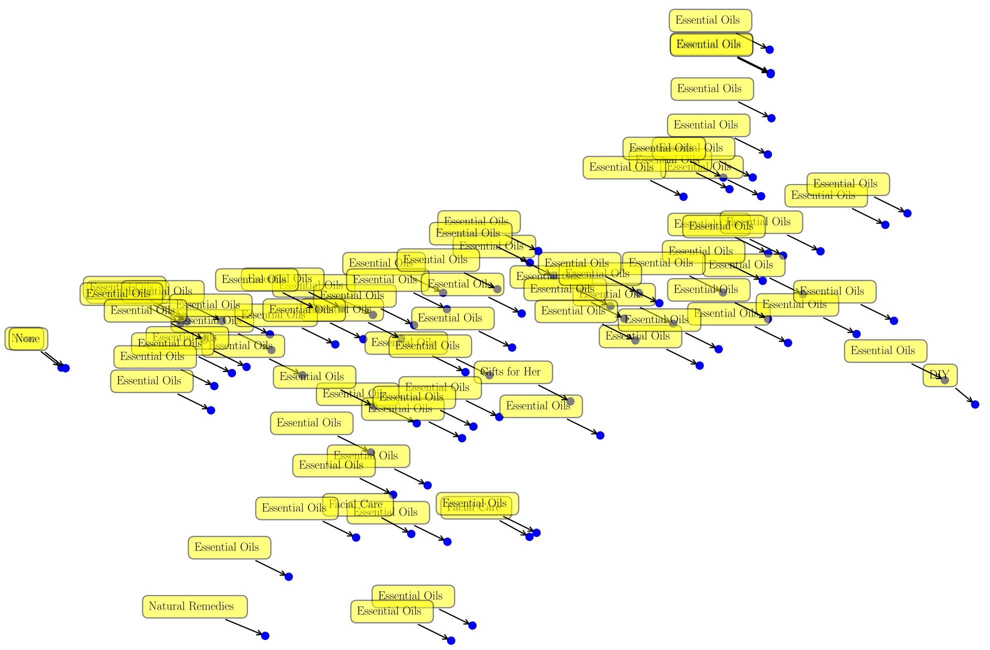


Figure 3.4: Red interaction cluster from Figure 3.2 labeled by category of product

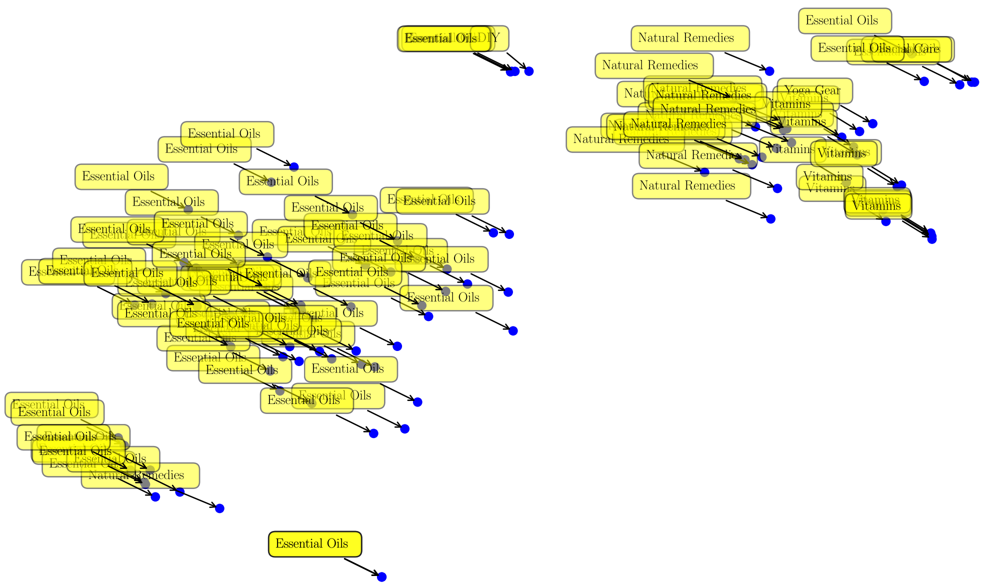
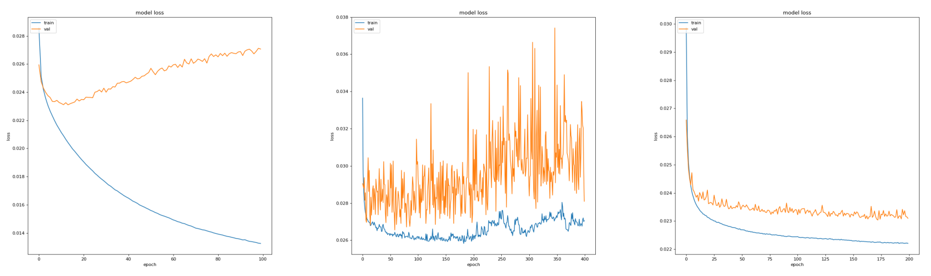


Figure 3.5: Red embedding cluster from Figure 3.3 labeled by category of product

validation errors for the various ways of regularization in Figure 3.6. Part (a) shows the training of the neural network without any regularization. You can see textbook overfitting after a few epochs. To (b) was added a dropout with the $\alpha = 0.5$ parameter specifying the number of omitted neurons, and the result is too much regularization that prevents learning. As promising, though unused in final, appeared L2 regularization with $\lambda = 10^{-6}$ illustrated in (c). Such a small λ value is chosen because the average output of training set is very close to zero.



(a) (b) (c)

Figure 3.6: No regularization (a), dropout of half of neurons (b) and L2 regularization (c). Validation error is orange, training error is blue.

As a final regularization methods were chosen batch normalization after each hidden layer with the cooperation of dropout of $\frac{1}{4}$ neurons.

3.5 Results

Now the promised results. *Recall* is always measured using the same users who were randomly selected from the interaction matrix. I compare *recall* of the three models:

Interaction recall uses the cosine similarity of *item's interaction vectors* as a metric of similarity between items. It is measured at the beginning of the experiment and does not change over time.

Embedding recall is the same, only items are represented by embeddings, not by interactions. It changes according to the embedding method and the used attributes.

Neural recall includes a trained neural network as a metric of similarity. It changes and is measured during network training. The table below lists the highest achieved and the appropriate *catalog coverage.*

The Tables 3.2 and 3.3 provide the final results of each measurement. The number in front of the arrow is *embedding recall* and *catalog coverage*,

Table 3.2: Results for dataset A

	Recall [%]	Catalog coverage [%]
Interaction	18.06	19.56
Doc2Vec	2.70 → 4.48	28.15 → 10.64
Hashing vectorizer	4.48 → 5.49	23.63 → 10.02
Bag-of-words	4.69 → 7.39	22.50 → 17.00
BoW, Sets	4.49 → 10.47	23.30 → 17.76
BoW, Sets, Numbers	3.91 → 12.38	21.47 → 17.83

Table 3.3: Results for dataset B

	Recall [%]	Catalog coverage [%]
Interaction	19.60	19.16
Doc2Vec	5.15 → 4.51	31.16 → 9.92
Hashing vectorizer	8.41 → 3.99	28.36 → 23.20
Bag-of-words	6.23 → 15.09	27.56 → 21.24
BoW, Sets	7.57 → 16.67	26.88 → 24.32
BoW, Sets, Numbers	8.40 → 18.47	27.56 → 21.20

after the arrow is the result of the network (*neural recall* and CC). From the word embeddings, Bag-of-words has the best results. In the case of the dataset B, it was even overwhelming, as the network barely learned to work with Doc2Vec and did not even cope with the Hashing Vectorizer. The BoW victory may seem very surprising, but it should be borne in mind that it is not only BoW, but embedding goes through other filters like tf-idf and especially LSA. In addition, the word embedding on the dataset B involves processing only a product name that is mostly unique, so Doc2Vec is not able to achieve the same quality as when processing full sentences in a native language. Adding additional attributes to embedding shows improvements in network performance. In the case of the dataset B, the embedding itself is more valuable for the recommendation. To the dataset A, the neural network must help with interpretation of embeddings. The biggest increase of *recall* can be seen on the last row of dataset A, when the network increase *recall* by 216%.

However, the greatest success of this method is the gained model trained using all the attributes from the dataset B. It reaches almost the same *recall* as the interaction and even goes beyond the interaction CC, which means that this model, created by proposed method, is able to recommend as well as the collaborative filtering, but it does not have a cold-start problem and can handle these recommendations for new products without interactions. Figure 3.7 shows the learning process along with the *recall*, and it can be expected that the resulting *neural recall* could still grow and possibly match the interaction one. The red and purple curves depict the evolution of training and validation error and are related to the left axis. On the right axis, you can see the values of *recall*. The *embedding* and *interaction recall* does not change during the learning process and is added to the graph only for illustration. Important is the evolution of *neural recall* shown in blue.

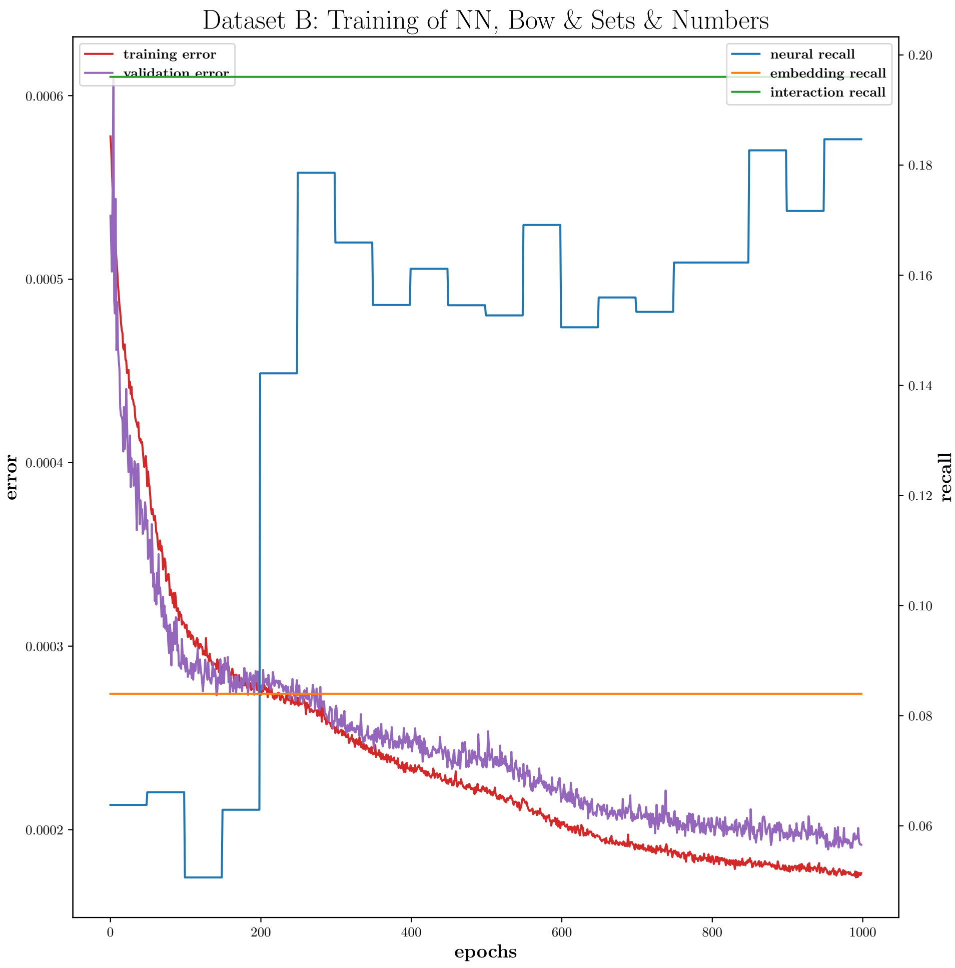


Figure 3.7: Training and validation error and recall during training of NN on dataset B using embedding of words, sets and numbers

For comparison, I supply Figure 3.8, which is very similar, but numbers are skipped, and only words and sets are embedded. It can be seen that the *neural recall* does not reach the results as in the case of Figure 3.7. In both cases, there is a tendency for moderate overfitting starting around the 200th epoch, but due to regularization, the validation error continues to fall, albeit not as quickly as a training error. Chart for each result from Tables 3.2 and 3.3 can be found in Appendix B.

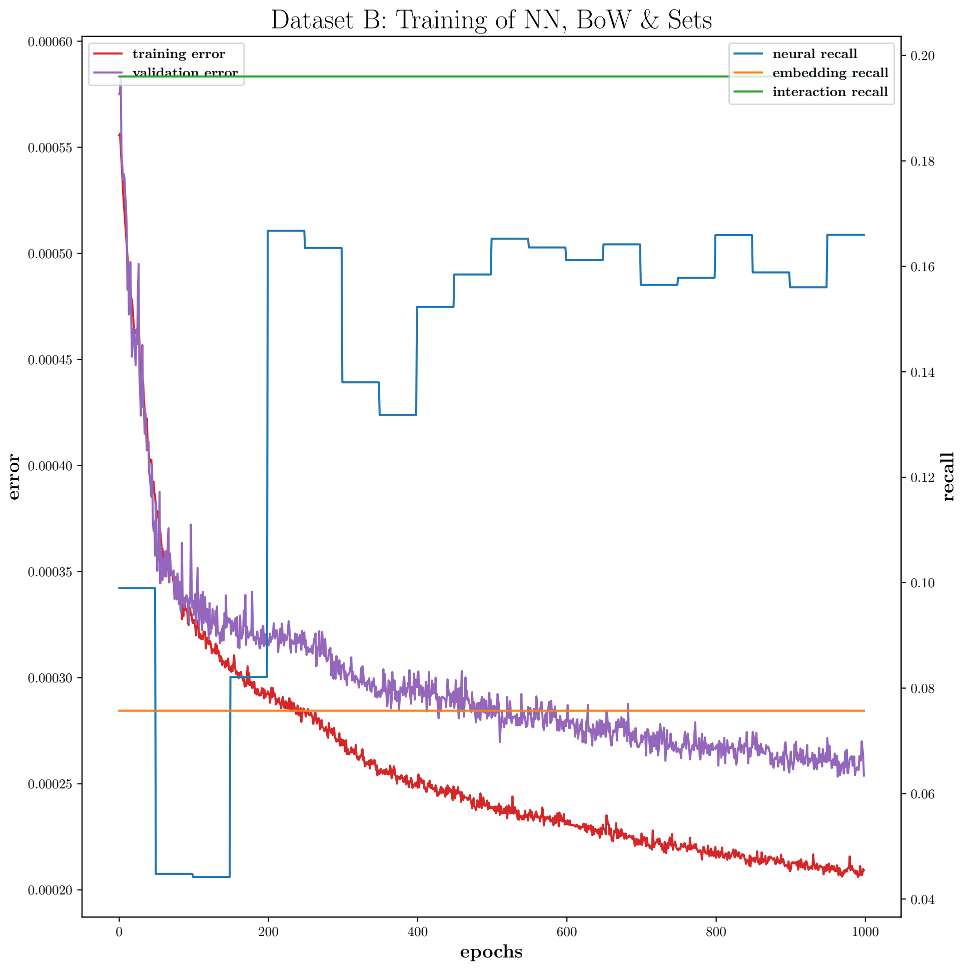


Figure 3.8: Training and validation error and recall during training of NN on dataset B using embedding of words and sets

Conclusion

This thesis aimed to design, implement and evaluate a new approach of the recommendation systems, which will be able to predict the interaction similarity based on the attributes of items. This goal has been fully achieved. Model, created by the proposed method, is able to give a more accurate recommendation by up to 216% than traditional content-based recommendations. In some cases the model matches the performance of collaborative filtering but does not require interactions and, therefore, solves the cold-start problem.

Different word embedding for processing of textual information about products has been compared within this method. The Bag-of-words model was chosen as the best (even better than Doc2Vec), and was used for the final embedding of product, which was later proceeded by the neural network. It was demonstrated that including more information about product leads to better recommendation. The proposed method can be extended to process, besides text, numbers, and sets, also images, videos, and others.

The work is designed to be a successful proof of the concept, and the developed method can process an utterly arbitrary dataset. Future work on this approach includes incorporating into complex recommendation engines and online evaluation using real users.

Bibliography

[1] Burke, R. Knowledge-based recommender systems. *Encyclopedia of Library and Information Science*, volume 69, 2000: p. 23. Available from: `https://pdfs.semanticscholar.org/1b50/ff4e5d8420f3ffae7de2a060b5a6fd4b8023.pdf`

[2] Mooney, R. J.; Roy, L. Content-based book recommending using learning for text categorization. In *Proceedings of the Fifth ACM Conference on Digital Libraries*, New York, NY, USA: ACM Press, 2000, ISBN 978-1-58113-231-1, pp. 195–204, doi:10.1145/336597.336662. Available from: `http://portal.acm.org/citation.cfm?doid=336597.336662`

[3] Tan, Z.; He, L. An Efficient Similarity Measure for User-Based Collaborative Filtering Recommender Systems Inspired by the Physical Resonance Principle. *IEEE Access*, volume 5, 2017: pp. 27211–27228, doi:10.1109/ACCESS.2017.2778424.

[4] Yuan, J.; Shalaby, W.; et al. Solving Cold-Start Problem in Large-scale Recommendation Engines: A Deep Learning Approach. *arXiv:1611.05480 [cs]*, Nov. 2016, arXiv: 1611.05480. Available from: `http://arxiv.org/pdf/1611.05480`

[5] Vasile, F.; Smirnova, E.; et al. Meta-Prod2Vec - Product Embeddings Using Side-Information for Recommendation. In *Proceedings of the 10th ACM Conference on Recommender Systems*, Boston, Massachusetts, USA: ACM Press, Sept. 2016, ISBN 978-1-4503-4035-9, p. 8, doi:10.1145/2959100.2959160. Available from: `https://dl.acm.org/citation.cfm?doid=2959100.2959160`

[6] Powers, D. M. W. Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. *Mach. Learn. Technol.*, 2008: p. 24. Available from: `https://www.researchgate.net/`

publication/228529307_Evaluation_From_Precision_Recall_and_F-Factor_to_ROC_Informedness_Markedness_Correlation

[7] Huang, T. S. Computer Vision: Evolution and Promise. 2018: p. 5. Available from: https://cds.cern.ch/record/400313/files/p21.pdf

[8] TensorFlow. Embeddings. [online], Mar. 2018, accessed on 2018-04-06. Available from: https://www.tensorflow.org/versions/master/programmers_guide/embedding

[9] Goldman, S. A. Embeddings | Machine Learning Crash Course. [online], Mar. 2018, accessed on 2018-04-05. Available from: https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture

[10] Maaten, L. v. d.; Hinton, G. Visualizing data using t-SNE. *Journal of machine learning research*, volume 9, 2008: pp. 2579–2605. Available from: http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf

[11] Wattenberg, M.; Viégas, F.; et al. How to Use t-SNE Effectively. *Distill*, volume 1, no. 10, Oct. 2016, ISSN 2476-0757, doi:10.23915/distill.00002. Available from: http://distill.pub/2016/misread-tsne

[12] Sahlgren, M. A brief history of word embeddings (and some clarifications). [online], Sept. 2015, accessed on 2018-04-06. Available from: https://www.linkedin.com/pulse/brief-history-word-embeddings-some-clarifications-magnus-sahlgren

[13] Harris, Z. S. Distributional Structure. *WORD*, volume 10, no. 2-3, Aug. 1954: pp. 146–162, ISSN 0043-7956, 2373-5112, doi:10.1080/00437956.1954.11659520. Available from: http://www.tandfonline.com/doi/full/10.1080/00437956.1954.11659520

[14] Mikolov, T.; Le, Q. Distributed Representations of Sentences and Documents. *CoRR*, 2014: p. 9. Available from: https://cs.stanford.edu/~quocle/paragraph_vector.pdf

[15] Ramos, J. Using TF-IDF to Determine Word Relevance in Document Queries. 2003: p. 4. Available from: https://pdfs.semanticscholar.org/b3bf/6373ff41a115197cb5b30e57830c16130c2c.pdf

[16] Brownlee, J. How to Prepare Text Data for Machine Learning with scikit-learn. [online], Sept. 2017, accessed on 2018-04-08. Available from: https://machinelearningmastery.com/prepare-text-data-machine-learning-scikit-learn/

[17] Mikolov, T.; Chen, K.; et al. Efficient Estimation of Word Representations in Vector Space. *CoRR*, Jan. 2013, arXiv: 1301.3781. Available from: `http://arxiv.org/pdf/1301.3781`

[18] Shperber, G. A gentle introduction to Doc2Vec. [online], July 2017, accessed on 2018-04-08. Available from: `https://medium.com/scaleabout/a-gentle-introduction-to-doc2vec-db3e8c0cce5e`

[19] McCormick, C. Google's trained Word2Vec model in Python · Chris McCormick. [online], Dec. 2016, accessed on 2018-04-09. Available from: `http://mccormickml.com/2016/04/12/googles-pretrained-word2vec-model-in-python/`

[20] Liu, H.; HUSSAIN, F.; et al. Discretization: An Enabling Technique. *Data Mining and Knowledge Discovery*, volume 6, 2002: pp. 393–423, ISSN 1384-5810, 1573-756X, doi:10.1023/A:101630430. Available from: `http://sci2s.ugr.es/keel/pdf/algorithm/articulo/liu1-2.pdf`

[21] Grbovic, M.; Radosavljevic, V.; et al. E-commerce in Your Inbox: Product Recommendations at Scale. *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015: pp. 1809–1818, ISSN 978-1-4503-3664-2, doi:10.1145/2783258.2788627. Available from: `http://doi.acm.org/10.1145/2783258.278862`

[22] Furnas, G.; Landauer, T.; et al. The Vocabulary Problem in Human-System Communication. *Commun. ACM*, volume 30, 1987: pp. 964–971, doi:10.1145/32206.32212.

[23] Dumais, S. Latent semantic analysis. *Annual Review of Information Science and Technology*, volume 38, 2004: pp. 188–230, doi:10.1002/aris.1440380105. Available from: `https://onlinelibrary.wiley.com/doi/abs/10.1002/aris.1440380105`

[24] Silva, I. N. d.; Spatti, D. H.; et al. *Artificial neural networks*. Switzerland: Springer International Publishing, first edition, 2017, ISBN 978-3-319-43161-1.

[25] Srivastava, N.; Hinton, G.; et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, volume 15, 2014: pp. 1929–1958. Available from: `http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf`

[26] Nowlan, S. J.; Hinton, G. E. Simplifying neural networks by soft weight-sharing. *Neural computation*, volume 4, 1992: pp. 473–493, doi:10.1162/neco.1992.4.4.473. Available from: `https://doi.org/10.1162/neco.1992.4.4.473`

[27] Nagpal, A. L1 and L2 Regularization Methods. [online], Oct. 2017, accessed on 2018-04-24. Available from: https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c

[28] Doukkali, F. Batch normalization in Neural Networks. [online], Oct. 2017, accessed on 2018-04-24. Available from: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c

[29] Ioffe, S.; Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167 [cs]*, Feb. 2015, arXiv: 1502.03167. Available from: http://arxiv.org/pdf/1502.03167

[30] Veen, F. v. The Neural Network Zoo. [online], Sept. 2016, accessed on 2018-05-10. Available from: http://www.asimovinstitute.org/neural-network-zoo/

Appendix A

Acronyms

ANN	Artificial neural network
BoW	Bag-of-words
CB	Content-based recommendation
CBoW	Continuous Bag-of-Words
CC	Catalog coverage
CF	Collaborative filtering
FFN	Feed Forward Network
HV	Hashing Vectorizer
LSA	Latent semantic analysis
MSE	Mean squared error
NN	Neural network
PV-DM	Distributed Memory version of the Paragraph Vector
RS	Recommendation system
tf-idf	term frequency–inverse document frequency

Appendix B

Figures

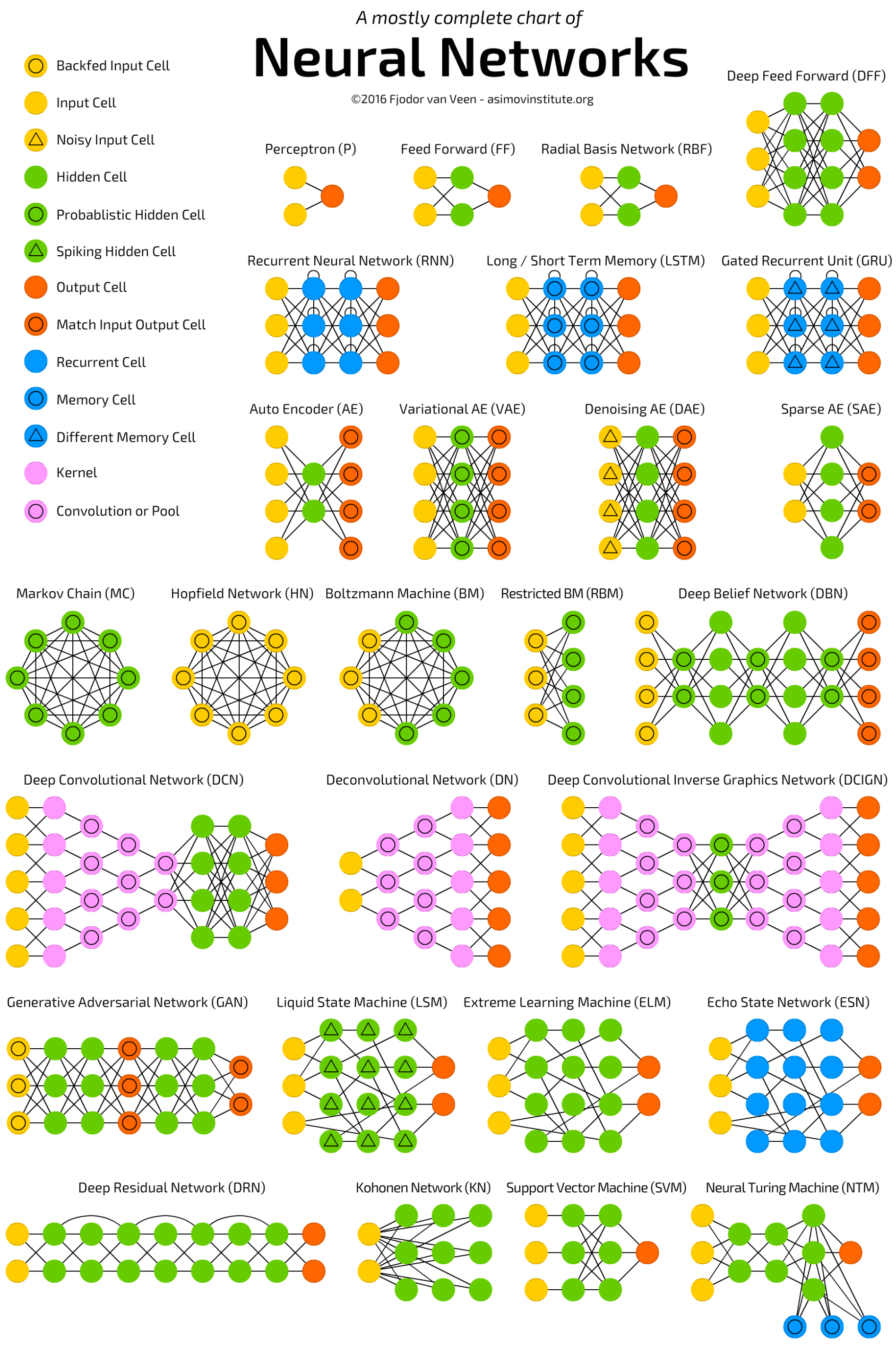

A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

Figure B.1: Architectures of Neural Networks [30]

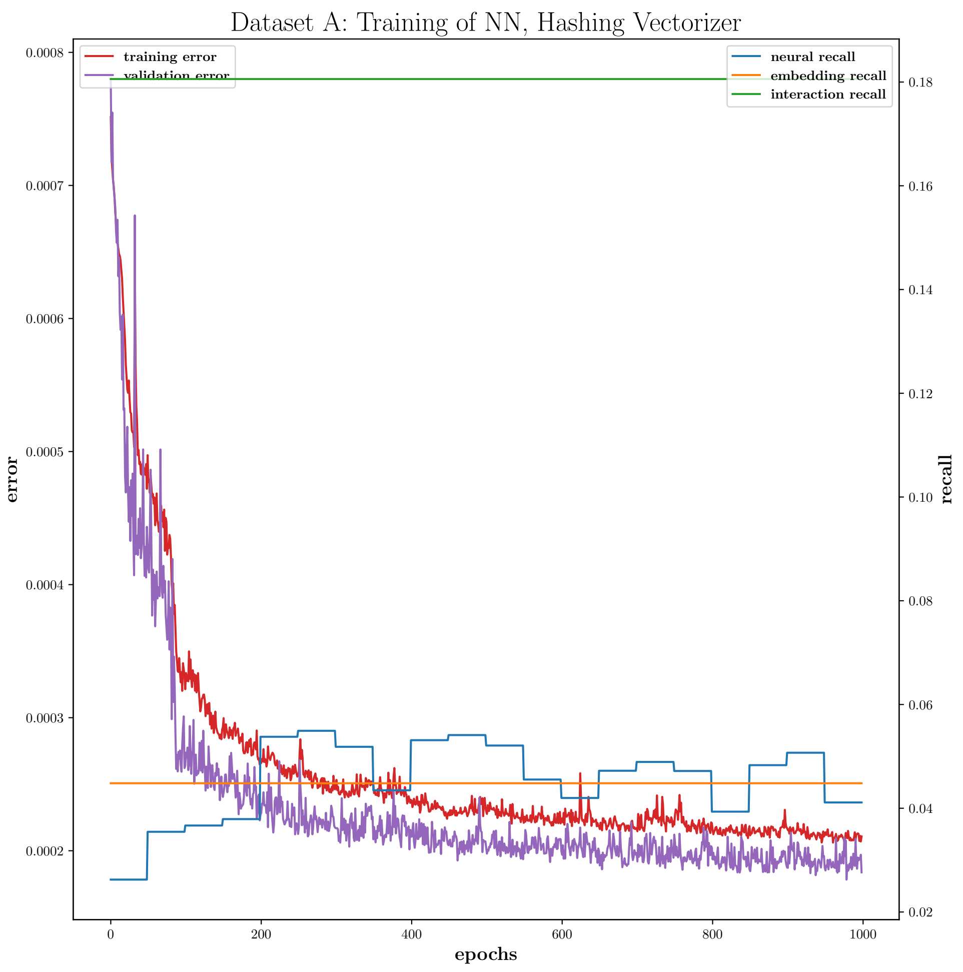


Figure B.2: Training of NN on dataset A from HV

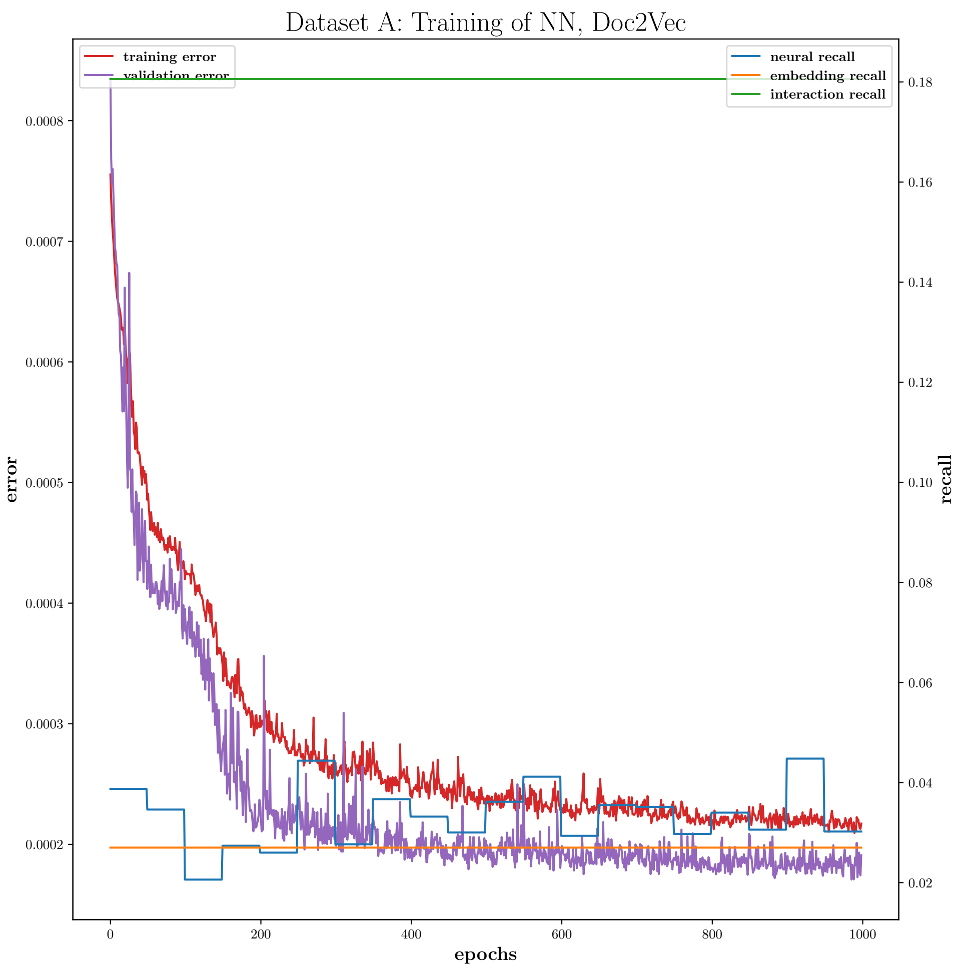


Figure B.3: Training of NN on dataset A from Doc2Vec

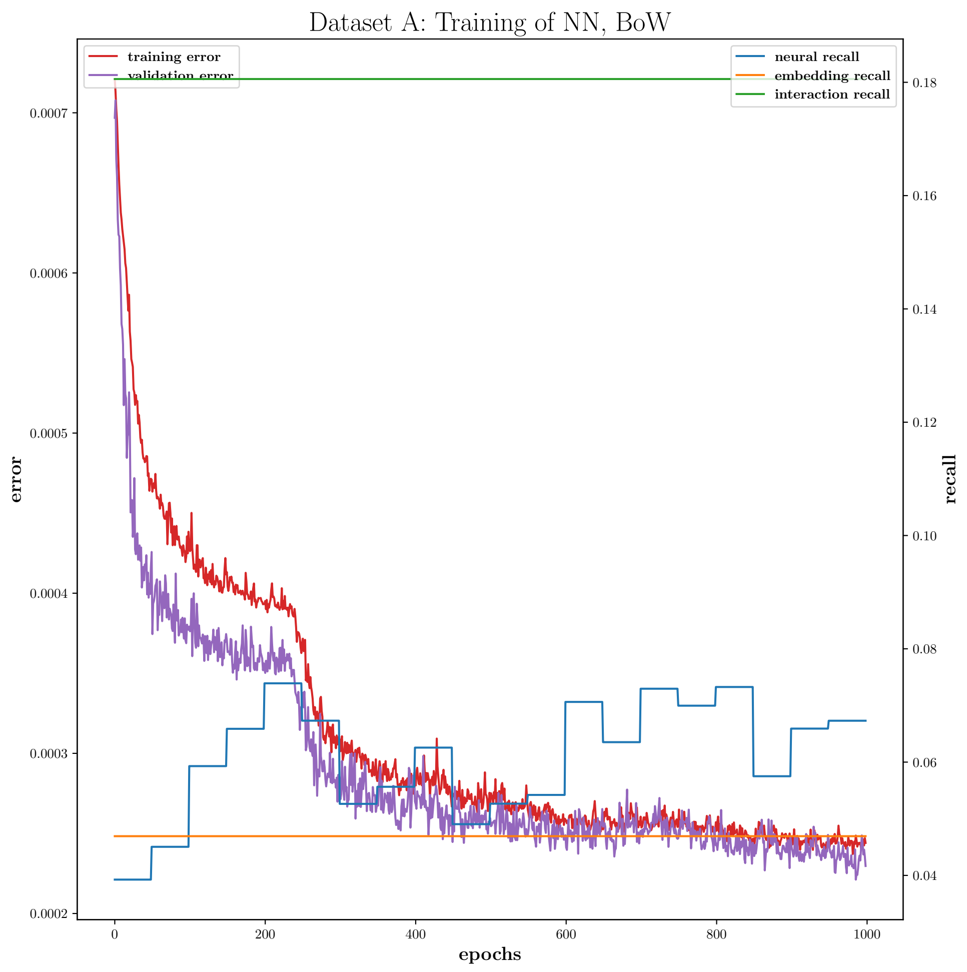


Figure B.4: Training of NN on dataset A from BoW

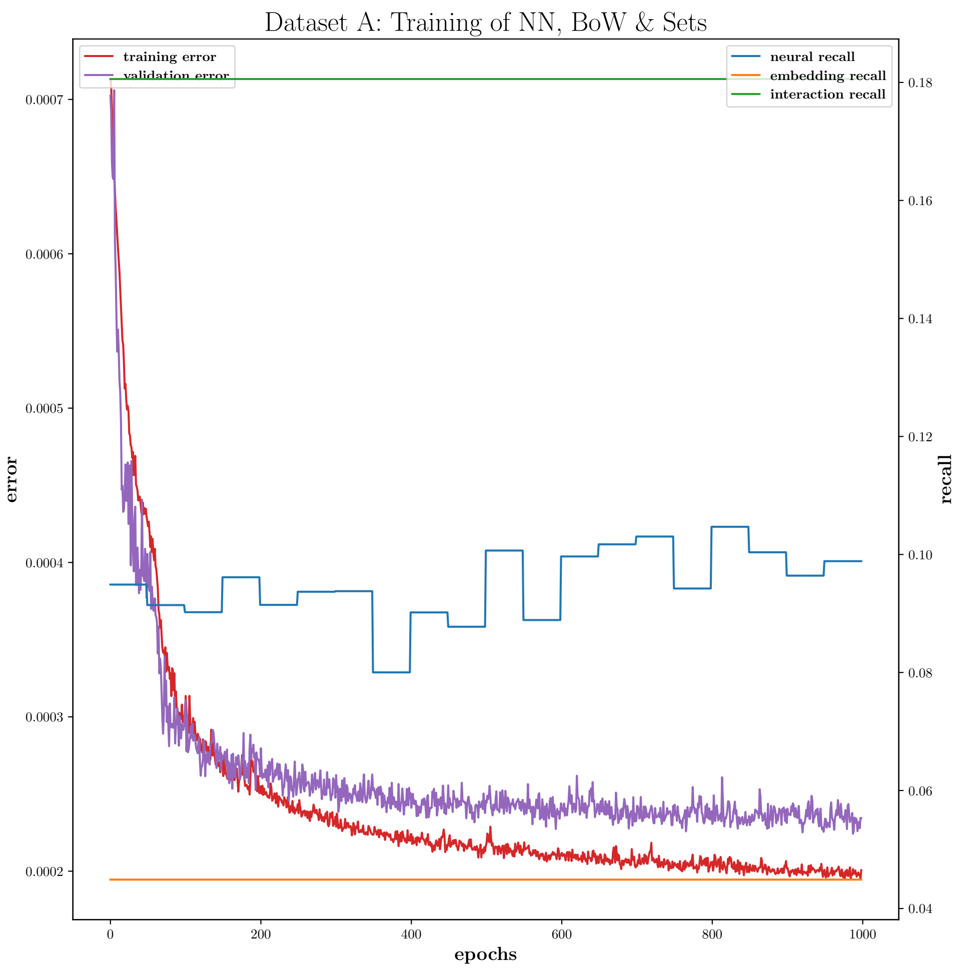


Figure B.5: Training of NN on dataset A from BoW and Sets

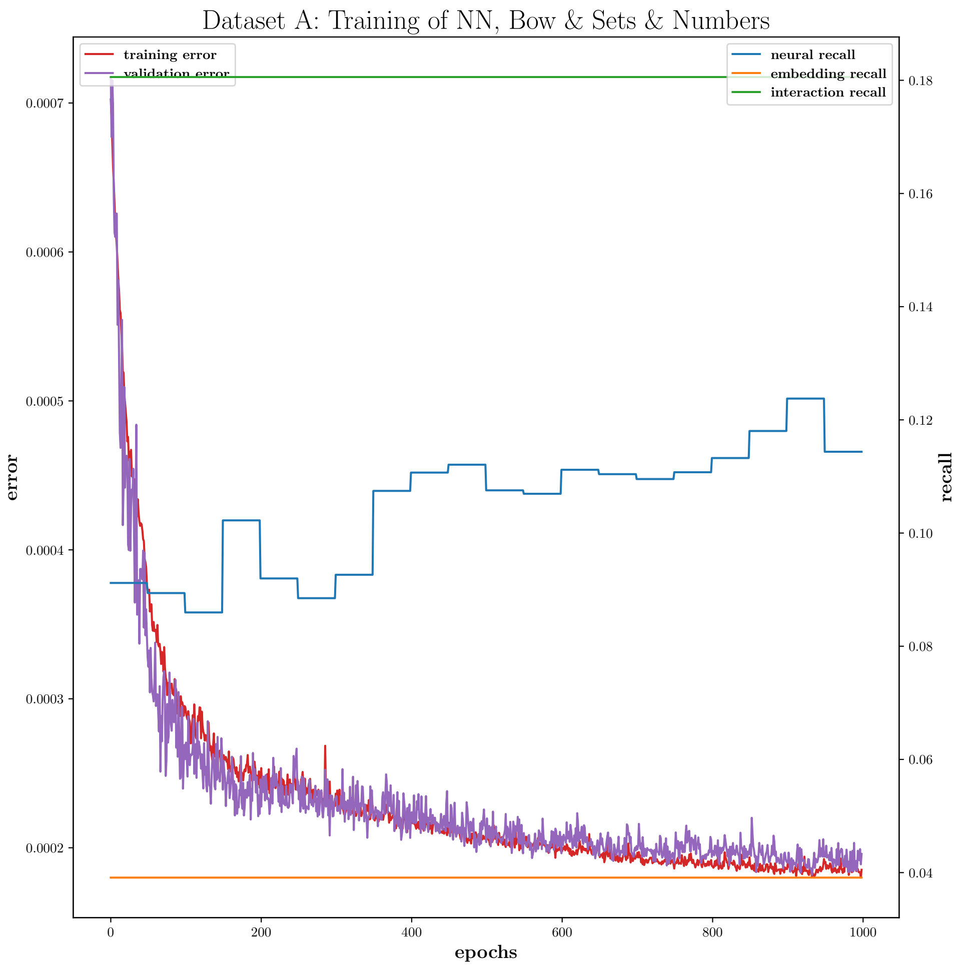


Figure B.6: Training of NN on dataset A from BoW, Sets and Numbers

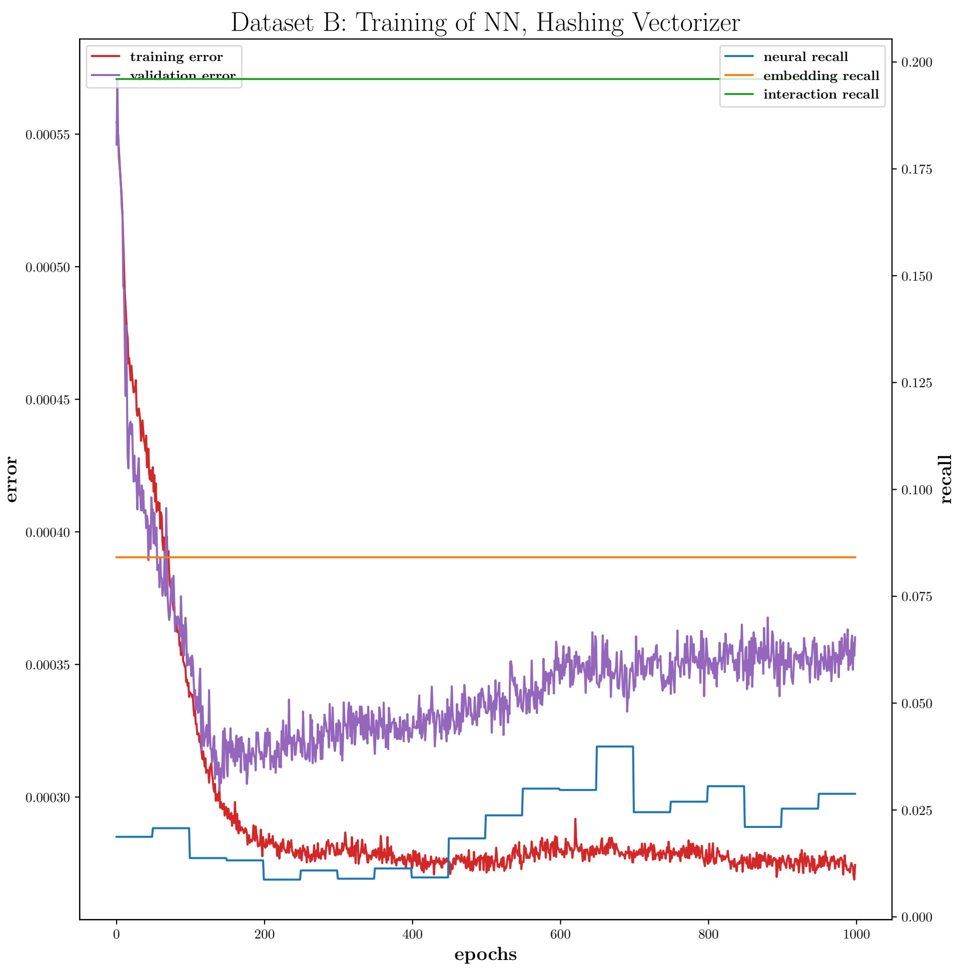


Figure B.7: Training of NN on dataset B from HV

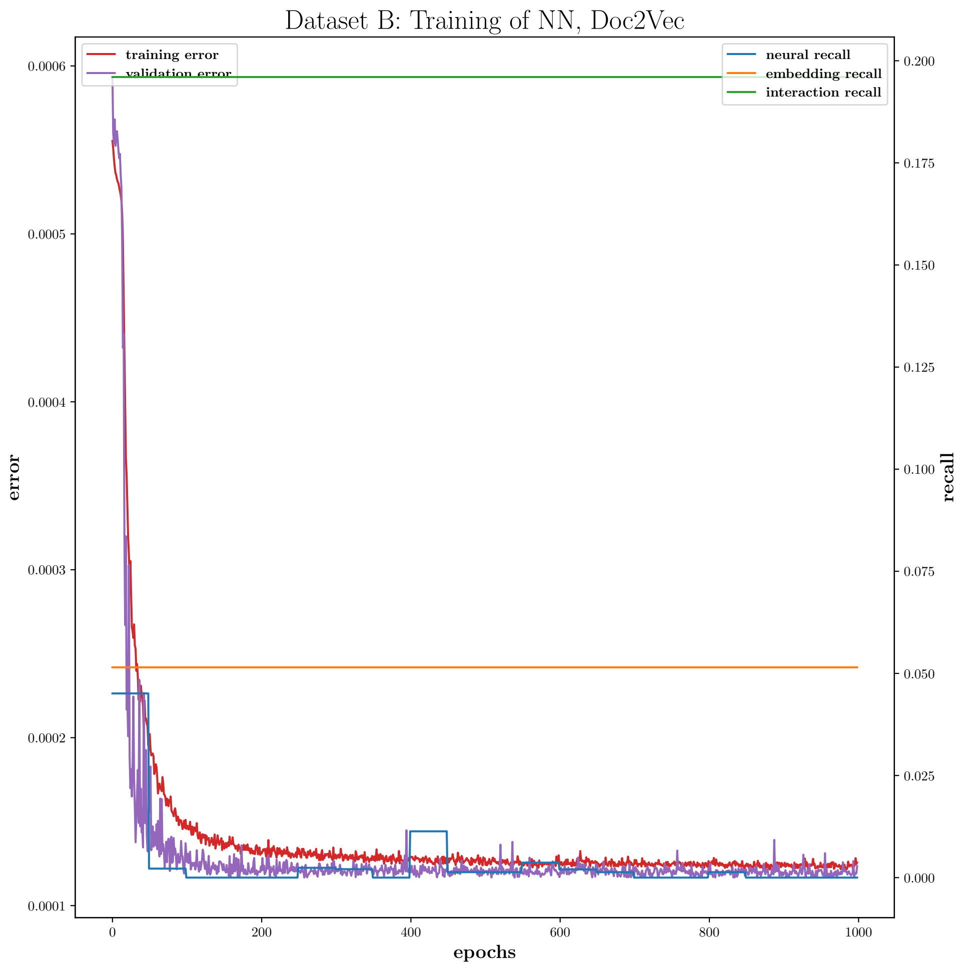


Figure B.8: Training of NN on dataset B from Doc2Vec

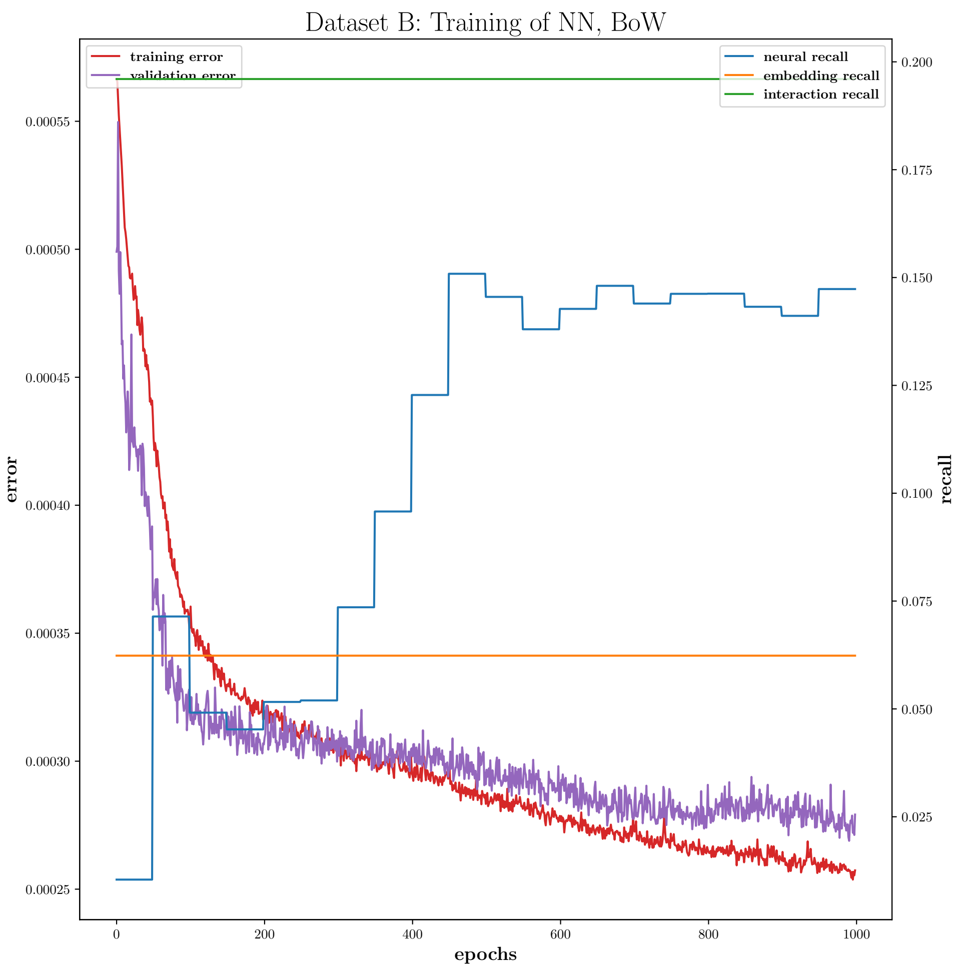


Figure B.9: Training of NN on dataset B from BoW

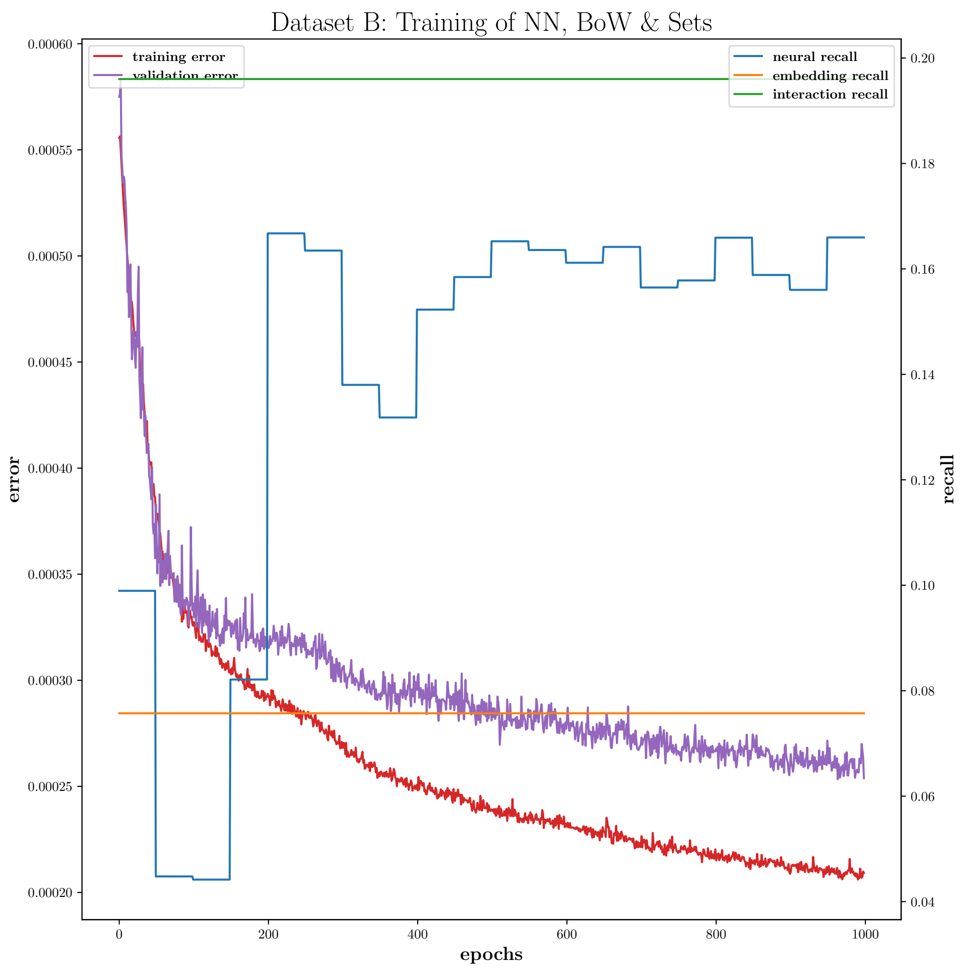


Figure B.10: Training of NN on dataset B from BoW and Sets

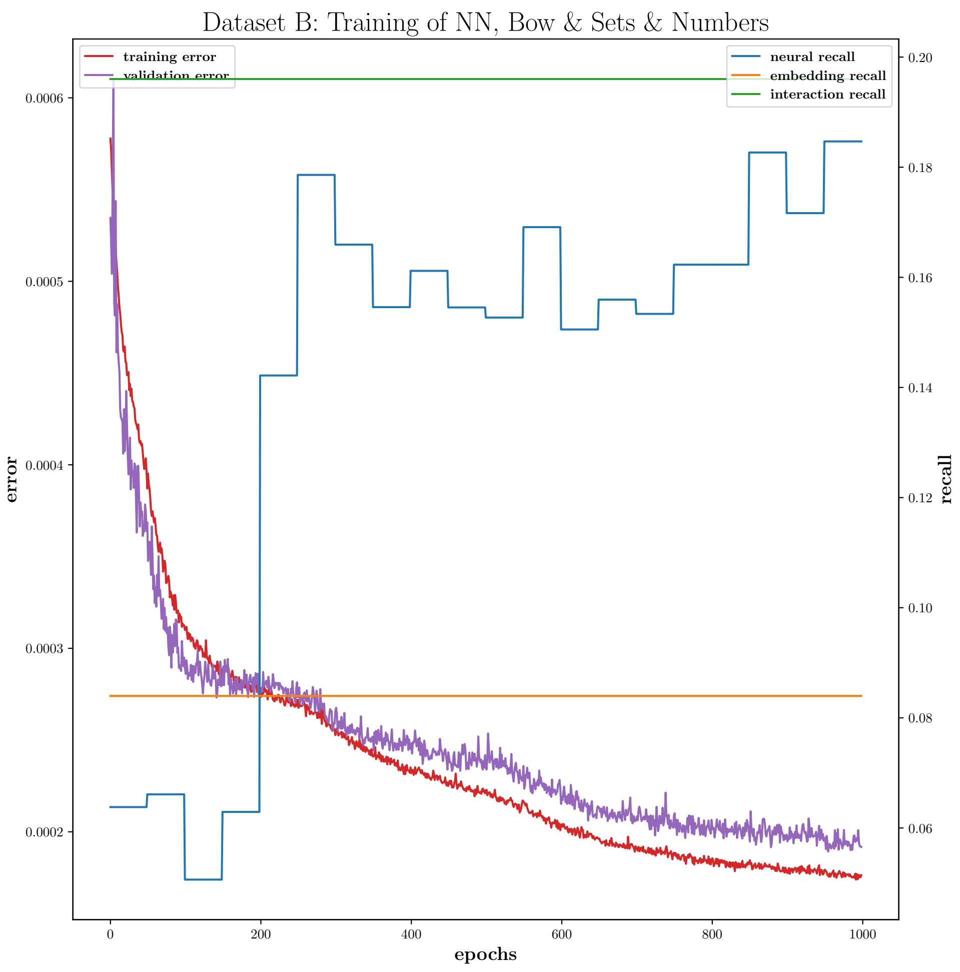


Figure B.11: Training of NN on dataset B from BoW, Sets and Numbers

Appendix C

Contents of enclosed CD

```
thesis.pdf ............................. the thesis text in PDF format
src .................................... the directory of source codes
   method.ipynb ................. implementation of proposed method
   settings.py .................................... dataset settings
   portuguese-stop-words.txt......... list of stop words for dataset B
   thesis ............. the directory of LaTeX source codes of the thesis
```