

UKRAINIAN CATHOLIC UNIVERSITY

MASTER THESIS

---

# Convolutional Graph Embeddings for article recommendation in Wikipedia

---

*Author:*  
Oleksii MOSKALENKO

*Supervisor:*  
Diego SÁEZ-TRUMPER

*A thesis submitted in fulfillment of the requirements  
for the degree of Master of Science*

*in the*

Department of Computer Sciences  
Faculty of Applied Sciences



APPLIED  
SCIENCES  
FACULTY

Lviv 2019

## Declaration of Authorship

I, Oleksii MOSKALENKO, declare that this thesis titled, “Convolutional Graph Embeddings for article recommendation in Wikipedia” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

---

Date:

---

UKRAINIAN CATHOLIC UNIVERSITY

*Abstract*

Faculty of Applied Sciences

Master of Science

**Convolutional Graph Embeddings for article recommendation in Wikipedia**

by Oleksii MOSKALENKO

In this master thesis, we were solving the task of a recommendation system to recommend articles to edit to Wikipedia contributors. Our system is built on top of articles' embeddings constructed by applying Graph Convolutional Network to the graph of Wikipedia articles. We outperformed embeddings generated from the text (via Doc2Vec model) by **47%** in Recall and **32%** in Mean Reciprocal Rank (MRR) score for English Wikipedia and by **62%** in Recall and **41%** in MRR for Ukrainian in the offline evaluation conducted on the history of previous users' editions. With the additional ranking model we were able to achieve total improvement on **68%** in Recall and **41%** in MRR on English edition of Wikipedia.

Graph Neural Networks are deep learning based methods aimed to solve typical Machine Learning tasks such as classification, clusterization or link prediction for structured data - Graphs - via message passing architecture. Due to the explosive success of Convolution Neural Networks (CNN) in the construction of highly expressive representations - similar ideas were recently projected onto GNN. Graph Convolutional Networks are GNNs that likewise CNNs allow sharing weights for convolutional filters across nodes in the graph. They demonstrated especially good performance on the task of Representation Learning via semi-supervised tasks as mentioned above classification or link-prediction.

## *Acknowledgements*

I would like to thank my supervisor Diego SÁEZ-TRUMPER, who helped me with this project a lot. Working under his supervision was a pleasure and a very unique and beneficial experience.

# Contents

<b>Declaration of Authorship</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>4</b>
2.1 Overview of Recommender Systems	4
2.1.1 Candidate Generation for RS (Filtering algorithms)	4
K Nearest Neighbors (kNN)	5
Performance issues of kNN and possible solutions	6
2.1.2 Ranking recommendation candidates	7
2.1.3 Metrics for results evaluation of Filtering and Ranking models	7
Filtering	7
Ranking	8
Other important metrics	8
2.1.4 Examples of production implementation of RS	9
YouTube	9
eBay	10
2.2 Overview of Representation Learning	10
2.2.1 Extracting features from text	10
word2vec	10
doc2vec	11
2.2.2 Structural representations	12
Matrix factorization	12
Random walk family	12
2.2.3 Deep Representation Learning	13
GraphSAGE	14
Aggregator functions for GraphSAGE	15
<b>3 Recommending Articles to Wikipedia Editors</b>	<b>16</b>
<b>4 Experiments and Evaluation</b>	<b>19</b>
4.1 Introducing Wikipedia dataset	19
4.1.1 Labels for Supervised learning	20
4.1.2 Text for document representation	21
4.1.3 Revision history for evaluation	21
4.2 Efficient (Big) Data Processing	22
4.2.1 Parallel Wikipedia Dumps pre-processing	22
4.2.2 GraphSAGE Scalable Implementation	23
4.3 Experiments	23

4.3.1	Document Representations . . . . .	23
4.3.2	GraphSAGE . . . . .	24
4.3.3	Deep Ranking . . . . .	26
4.3.4	Time optimization for kNN search . . . . .	26
4.4	Evaluation . . . . .	27
4.4.1	Result Interpretation . . . . .	27
4.5	Results . . . . .	29
<b>5</b>	<b>Conclusion</b>	<b>31</b>
	<b>Bibliography</b>	<b>33</b>

*To my patient wife*

## Chapter 1

# Introduction

Recommendation systems (RS) have been a very popular topic in Machine Learning for a long time. It is also one of the most practically applied areas since the benefits of recommending a product to a customer, or a new movie to a viewer are obvious.

The explosive growth of the amount of information and services provided through the internet raised even bigger demand for RS. Variety of choice does not always mean good offering. Users now need assistance in making such choice and RS became such assistant. A large number of RS is built on the assumption that user will more likely choose items similar to his preferences (previous items he interacted with) [Ricci, Rokach, and Shapira, 2015]. The quality of similarity measurement is a very important factor affecting RS performance [Bobadilla et al., 2013]. Recommender system algorithms are utilizing the concept of similarity (or distance) between objects (item-based recommendations) and/or between users (user-based recommendations). For example, to recommend some product, we can find the most similar item to what user bought before, or to recommend some movie we can find a user profile with similar preferences (history of views) and pick something for his history.

For some cases, this task of similarity is relatively easy when our object possesses quantitative features (ex. a movie rating) since the notion of distance is easier to apply in such setup. But in most cases, when we compare images or texts, it is necessary to have a separate layer responsible for mapping those objects (ex. images) to a low-dimensional numerical space. This topic has been actively researching lately especially with the growing popularity of Deep Learning approaches and their applications to RS [Zhang, Yao, and Sun, 2017].

With rising popularity of Neural Networks and particularly Convolutional Neural Networks (CNNs) it became much easier to compare some of high-dimensional objects (like images or text) by producing *Embeddings* - low-dimensional vectors that represent the original object with preservation of as many as possible of its properties (ex. semantic properties: embeddings of words with similar meanings must be located close in the vectorial space). However, images have a unique property - some natural order - pixels are ordered in rows and columns - that made the task for embedding producing simpler. If we will decide to represent a more complex-structured object, like Graphs, the problem became more complicated and, in fact, it is not completely solved yet [Hamilton, Ying, and Leskovec, 2017b]. However, some impressive results were already achieved by applying ideas from CNNs to Graphs. Graph Convolutional Networks [Duvenaud et al., 2015; Kipf and Welling, 2016] is a family of Neural Network models that have a common idea to collect knowledge



about the neighborhood for each node by passing the data (state) from one node to another through some filters (convolutions) that are shared across the full Graph. It allows to generate unique *Embeddings* for each node in the Graph that will combine structural knowledge with original node features. Further studies proposed to observe in a moment of time not a full graph but only some sample of it, which made it possible to work with Graphs of massive scale without losing Embedding's quality [Hamilton, Ying, and Leskovec, 2017a; Chen and Zhu, 2018].

**In the present master's thesis** we want to explore the task of Representation Learning by applying Graph Convolutional Networks to the Graph of Wikipedia Articles. Our goal is to create a RS to suggest articles to editors of Wikipedia based on constructed articles' embeddings. Analogous with product recommendation, in this setup articles act as products and editors as users.

The Wikipedia projects receive more than 1 million editions per day. From writing full articles to focus on grammatical fixes in multiple articles, editors contribute in diverse ways. Therefore, recommending a new task for an active editor is not trivial. In this project, we want to create a Recommender System, that understands editors behavior, and recommend them new tasks.

One of our challenges is to address scalability issues caused by the size of the given dataset (e.g. English Wikipedia has more than 6M of articles and over 500M of links connecting them). We will show that this issue could be transformed into a benefit with carefully selected architecture for the system. We will conduct experiments and compare our solution of embeddings generation with representations received from the text.

The main contributions of the present work can be categorized in the following groups:

- **We propose and implement a GCN-based solution to represent articles and users in Wikipedia.** Our approach uses link-prediction model, that improves article-embedding quality, compared with pure-text based models such as Doc2Vec. To the best of our knowledge, this is the first work applying that technique in the Wikipedia context.
- **The implementation of an scalable Recommender System** for Wikipedia contributors, using the state-of-art techniques on this field. Our model is able to efficiently deal with huge data, in a *real-world scenario*, such as the full English Wikipedia eco-system. Our model is able to work on-line, without requiring full re-training when new users or articles are added. And it also implements an efficient Nearest Neighbors approach, to produce fast and reliable recommendations in real-time scenarios.
- **We publicly release all the code developed in this work.** Two main packages are shared: one, for efficiently processing Wikipedia Dumps using Spark, and another, with an scalable implementation of one of the state-of-art Graph embeddings algorithms [Hamilton, Ying, and Leskovec, 2017a]. While original implementation failed with relative medium-size graphs (100M of edges), our implementation is able to manage the full English Wikipedia Graph, with more than 450M edges.

The reminder of this work is organized as follows: In Chapter 2, a brief overview of modern Recommender System architecture is given. We provide a theoretical background overview for our Representation Learning Model and give some real-world examples of built RS with billions of items and millions of users; In Chapter 3 we describe our solution for the task of recommending articles to editors of Wikipedia based on latest researches in the area; In Chapter 4 our experimentation setup, input data and our preprocessing pipelines are described. Pre-processing is another big challenge due to dataset size, so we describe our contribution aimed to tackle that problem. Results of the offline evaluation of the built system based on the history of previous user contributions from Wikipedia are provided. We show that graph-based representations give significant improvement in comparison with our baseline Doc2Vec representations even in unsupervised setup; Finally, in Chapter 5 we present our conclusions and summarize the main outputs of this work.

## Chapter 2

# Background

### 2.1 Overview of Recommender Systems

It is hard to imagine a modern high-tech user service: website or mobile application, that does not leverage from applying Recommender System (RS). We can recall the obvious cases like recommending a movie or book but it's much more than that - RSs are everywhere: suggesting product to buy (e-commerce), news to read (media), people to add friend/follow (social), most of the search systems are personalized and results are ranked by an RS. It brings to an endless ocean of information some personification and relevance for every specific user. This is a well-known way to improve user's experience. But despite the fact that RS has to work with completely different kind of data: images, texts, videos, website pages - we can formulate common features that will be applicable for most of the mentioned cases:

- **Candidate Generation:** The filtering algorithm is being used to pick only some subset of an items from the available database that are relevant for the specific user.
- **Ranking:** The algorithm that is applied to sort recommendations from the most relevant items to the least.

Recently, there has been a lot of new researches regarding new filtering and ranking algorithms especially with the growing popularity of Deep Learning approaches [Zhang, Yao, and Sun, 2017]. This is also very intensively developing area due to an exponentially growing amount of data on the internet. In the next parts, we will explore those algorithms in details.

#### 2.1.1 Candidate Generation for RS (Filtering algorithms)

This task consists in looking up in the database of known items and return only the relevant ones to the given *query*. In the recommendation setup query may be replaced with user representation, which is in most cases built based on his history of interactions (purchases/likes/article editions) along with personal features (age, sex, etc.).

We can group most of the known filtering algorithms for candidate generation into the next categories: collaborative; demographics; social; content-based and hybrid.

**Collaborative filtering (CF)** is based on assumptions that we will most probably like things that recommended by people with similar preferences to ours. The simplest collaborative algorithm can be formulated as: find other users that previously gave a score (or bought/read) to the same items that we did and pick the items from their history that we do not know about (user-user collaboration). However, many other similarity measures between users could be used. Several approaches like to group users by personal attributes (like address, age) or by social connections are distinguished to separated categories like **Demographic** and **Social filtering** respectively. That could be explained by the fact that classic Collaborative filtering has a weakness named *Cold-Start*. In order to get proper recommendation user has to have filled history of interactions (bought/read) with products for RS to be able to compare it with other users. It is especially tangible limitation of RS since it has to interact with new users all the time. Similarly, new products that do not have scores or purchases are less likely to be recommended. Demographic and Social filtering are aimed to solve the first part of the problem. Whereas **Content-based filtering** can help with the second and partly the first - user does not have to have a long history of interactions, we can start to recommend something after the first contact. Content-based filtering is utilizing similarity between objects (products) to pick new items similar to ones that user already bought/read/scored positively. **Hybrid filtering** implies a combination between already mentioned approaches like CF plus Demographic filtering or CF plus Content-based with the aim to address different solutions for Cold-Start problem.

### K Nearest Neighbors (kNN)

In all of the cases described above, when the similarity between items or users is implied *k Nearest Neighbors* (kNN) algorithm could be applied as a core of filtering model. There are many different implementations such as: *exact kNN* which implies calculating distance from *query* (q) to all items in database and then sort from the closest to furthest ones and pick first *k*; *approximate kNN* like Locality-Sensitive Hashing [Liu et al., 2004] which balance between accuracy and speed; *index-based solutions* like Inverted multi-indexes [Babenko and Lempitsky, 2012] aimed to solve scalability issues of classic kNN. Approximate kNN solutions combine high recommendation performance with computation efficiency and are created to be used in real-world tasks when latency is critical and size of database is significantly exceed the amount of RAM.

For kNN algorithm to work, it has to have a measure of distance between items. Among the most commonly used metrics we have: *Pearson Correlation*, *Cosine*, *Euclidean*, *Jaccard distances*. Selection of the metric fully depends on the characteristics of the target object. In most of the cases when representations were learned with Deep Learning approach resulted vectors could be compared via cosine similarity.

$$similarity = \frac{A \cdot B}{\|A\| \|B\|} \quad (2.1)$$

$$distance = 1 - \frac{A \cdot B}{\|A\| \|B\|} \quad (2.2)$$

## Performance issues of kNN and possible solutions

Described kNN search could be reformulated as Maximum Inner-Product Search (MIPS) problem, which due to the involvement of matrix multiplication in straightforward solution does not guarantee a good speed and computational efficiency with growing amount of search area. Optimizing MIPS problem - is currently an active research area. Local Sensitive Hashing - is a popular approximate solution to this problem. This method reformulates the problem into querying objects with distance no more than  $(1 + \epsilon)$  times bigger than the distance of true  $k^{th}$  nearest-neighbor and considerably improve the query time in return [Liu et al., 2004].

LSH is providing a projection of input vectors into small alphabet with a set of trainable hash functions. It is based on the idea that if two points are close they will be mapped to the same value in at least some dimensions (fall to the same bucket) with high probability. And the opposite, if the distance is too big - they will not share common buckets. However, initially, algorithm was developed for applying on Euclidean space with respectively Euclidean distance, which is not compatible with Cosine distance [Gionis, Indyk, and Motwani, 1999]. The solution for that involves a transformation from Cosine into Euclidean space. One of such method is described in Bachrach et al., 2014 and is implemented in well-known C++ library Annoy by Spotify<sup>1</sup>.

Another approximate solution is Hierarchical Navigable Small World [Malkov and Yashunin, 2016]. It involves building a graph where vectors are represented by vertices and edges denote neighborhood relations between vectors: short edges denote small distance between vectors, long - create "Small World". It is based on the idea that in the "Small World" average path between two disconnected vertices will take  $\log N$  steps, where  $N$  - is amount of vertices. All vectors are being sampled into nested subsets (layers). From the smallest subset (with just few vector) layer size increases following geometrical progression. On each layer graph with edges based on distances between vectors is built. All vectors that were present on smallest layer are always exist on the next one. The search starts on the smallest layer, the shortest edge is always chosen to the next vertex, and then moves to next layer, where detected neighbor serves as transition point from one layer to another. Procedure repeats until we reach the biggest layer, where breadth first search is producing resulting nearest neighbors set.

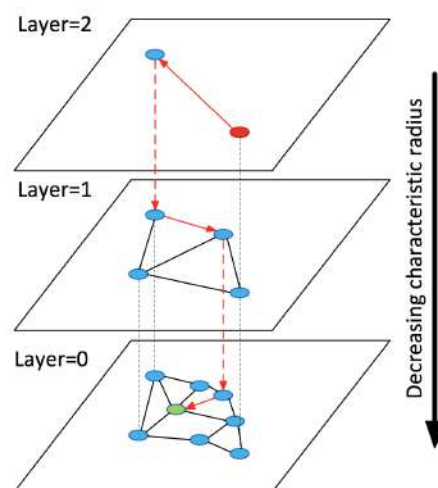


FIGURE 2.1: Illustration of the Hierarchical NSW idea. The search starts from an element from the top layer (shown red). Red arrows show direction of the greedy algorithm from the entry point to the query (shown green) [Malkov and Yashunin, 2016]

<sup>1</sup><https://github.com/spotify/annoy>

### 2.1.2 Ranking recommendation candidates

After we selected a subset of relevant objects from the database we can look at them more carefully to sort them in best possible ordering. The best possible ordering here implies that items that user will more likely to buy/watch/edit should be placed at the beginning of the recommendation list.

First, since we already have a measure of distance and we calculated it on the first step of *exact kNN search* - we can use those distances as inverted rank. But this approach has a huge weakness of so-called hubness. Hubness is the limitation of vector space data when some objects (hubs) are much more likely to occur in different kNN searches despite the fact that they are not always relevant to the query but just positioned close to many other vectors (centrality) due to skewness in data distribution. This especially affects word2vec and doc2vec models [Dinu and Baroni, 2014].

As an alternative, a separate model could be trained to predict the rank of the object in the result list. Ranking algorithms could be categorized in next groups according to [Liu, 2009]: **pointwise models**, when final score is being predicted only from the query and one (at a time) recommended item features; **pairwise models**, when items in the recommended list are compared to each other, which could be simplified to binary classification problem:  $rank(a) \leq rank(b) = \{0; 1\}$ ; **listwise models** are trying to predict all ranks simultaneously by maximizing objective metric that we will discuss below.

As the example of pointwise approach let us consider the model in a form of binary classification task. We can label with class 1 all items  $i$  for query  $q$  that were relevant (was actually selected by user) recommendations. Similarly, all non-relevant recommendations will have class 0. At serving time items with higher probability to have class 1 will be moved to the start positions in the list. The model thus became a Logistic Regression task [Gey, 1994].

$$Pr(Relevance|item) = \frac{1}{1 + e^{-c - W[x_i, q]}} \quad (2.3)$$

where  $W$  and  $c$  are learnable parameters trained on verified relevant recommendations and  $[x_i, q]$  - concatenation of feature vectors of item and query. In the case when simple linear regression is not enough non-linear Neural Network proved to be efficient [Covington, Adams, and Sargin, 2016]. The ranking part in recommendation system for YouTube videos uses this approach but with several fully-connected layers with ReLU to handle wide features from embeddings.

### 2.1.3 Metrics for results evaluation of Filtering and Ranking models

#### Filtering

It is important to verify that selected items are aligned with actual users' preferences. Among others metrics we are finding worth to mentioned: precision, which shows the fraction of relevant recommendations among all recommended items; and recall,

which indicates how many items were recommended among all relevant ones.

$$precision = \frac{1}{|U|} \sum_{u \in U} \frac{|\{i \in Z_u | r_{u,i} > \theta\}|}{K} \quad (2.4)$$

$$recall = \frac{1}{|U|} \sum_{u \in U} \frac{|\{i \in Z_u | r_{u,i} > \theta\}|}{|\{i \in Z_u | r_{u,i} > \theta\}| + |\{i \in Z_u^c | r_{u,i} > \theta\}|} \quad (2.5)$$

where  $U$  - set of all users  $u$ ,  $Z_u$  - set of recommended items,  $Z_u^c$  - converse set to  $Z_u$ ,  $r_{u,i}$  - recommendation score for user  $u$  and item  $i$ ,  $\theta$  - recommendation threshold.

## Ranking

Quality of ranking model could be measured with: *discounted cumulative gain* (2.6), which adds to correct recommendation logarithmic gain if it is located far from the beginning of the recommendation list; *mean reciprocal rank* (2.7), which indicates the average position of relevant recommendation in the list.

$$DCG = \frac{1}{|U|} \sum_{u \in U} (r_{u,p_1} + \sum_{i=2}^K \frac{r_{u,p_i}}{\log_2 i}) \quad (2.6)$$

$$MRR = \frac{1}{|U|} \frac{1}{|Z_u|} \sum_{u \in U} \sum_{p_i \in Z_u'} \frac{1}{i} \quad (2.7)$$

where  $p_1, \dots, p_K$  is list of recommendations,  $r_{u,p_i}$  - is ground truth rating given by user  $u$  to item  $p_i$ ,  $Z_u'$  - set of correctly recommended items.

## Other important metrics

We should mention no less important measurers of recommendation quality as *Novelty* (4.8) and *Diversity* (4.9) [Bobadilla, Serradilla, and Bernal, 2010]. It is sometimes a false assumption that user seeks for only high ratable or the most similar recommendations [McNee, Riedl, and Konstan, 2006]. Recommendation must not only bring to users completely identical or very close items but in the same time try to introduce something unexpected that will potentially satisfy them.

$$novelty_i = \frac{1}{Z_u - 1} \sum_{j \in Z_u} 1 - similarity(i, j) \quad (2.8)$$

$$diversity_{Z_u} = \frac{1}{|Z_u|(|Z_u| - 1)} \sum_{i \in Z_u} \sum_{j \in Z_u, i \neq j} 1 - similarity(i, j) \quad (2.9)$$

We also think, that those metrics can be applied to measure the user itself. We can say a lot about his preferences by measuring diversity of his history of interactions.



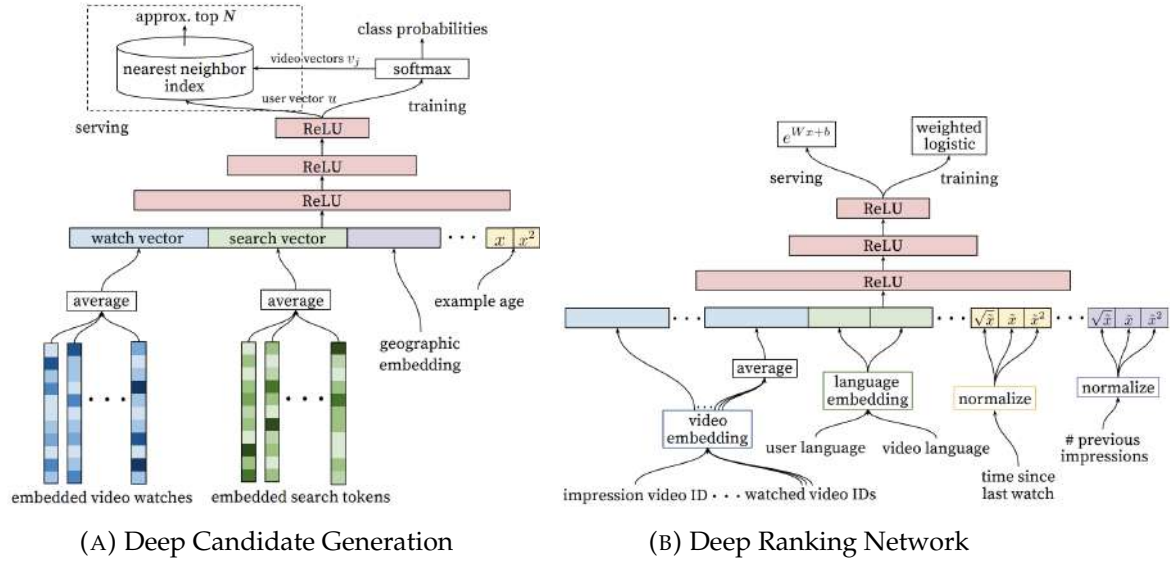


FIGURE 2.2: Architecture of retrieval: (A) and ranking (B) parts of YouTube recommender system with applying Deep Learning models [Covington, Adams, and Sargin, 2016]

### 2.1.4 Examples of production implementation of RS

#### YouTube

One of the main features of YouTube - the world largest platform for creating and discovering video content - is to deliver highly personalized recommendations. This system was built according to the classic two-stage information retrieval design: 1. candidate generation (search in the database of existing items); 2. ranking model, which sorts candidates by the probability of user to spend on item expected watch time and returns only  $K$  most probably watched. Both models share similar Deep Learning architecture (Figure 2.2) [Covington, Adams, and Sargin, 2016].

Candidate Generation model takes as input history of video watches, search tokens and some additional features. Video watches encoded as video ids (from the database) are being converted into dense low-dimensional embeddings. Those embeddings are learned jointly with the model itself. After passing through several fully-connected layers with ReLU activation model produces embeddings of the user. It is being learned on the task of multiclass classification when the model tries to predict which exactly video (class) the user will watch at a time  $t$  based on user and context features. Produced user and video embeddings are then used to generate candidates by Nearest Neighbors search.

Then the ranking model is trained to predict watch time for each video impression. It is done via training Deep Network on weighted logistic regression task when watch times from history are being attached as weights to positive examples (impression was clicked), negative - receive unit weight. Cross-entropy loss is being applied on training step.



## eBay

A similar approach is used for recommending items on eBay (Figure 2.3): candidates are being generated on the first stage (Recall) according with given query (Seed) and then being ranked by the pointwise model with items having the highest probability to purchase go first to return. Recall system is utilizing simple and reliable TF-IDF similarity (implemented in ElasticSearch) and users coviews (pair of items which have been frequently viewed together in the same browsing session by multiple users). Ranking model is optimized on similar to YouTube binary classification task when positive class is clicking/purchasing and negative - lack of action [Brovman et al., 2016].

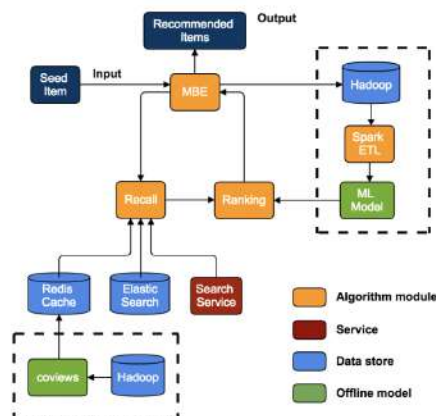


FIGURE 2.3: Architecture of Merchandise Backend (MBE) system for recommendation items on eBay [Brovman et al., 2016]

## 2.2 Overview of Representation Learning

### 2.2.1 Extracting features from text

#### word2vec

One of the fundamental problems in Machine Learning and in particular crucial for similarity task - is to find a good representation of real-world object in machine accessible format. Since machines are operating exclusive with numbers only - the natural choice for the format would be representation in vector space. And to control computational complexity this vector space must be of reasonably low dimension. The definition of good representation is various from task to task but what is common - that we would like to preserve as many properties of the original object as possible. In Natural Language Processing problems top priority task is to teach machine to understand the meaning of texts, sentences and words to be able to operate with those meanings - for tasks like paraphrasing or question answering. It is called Semantic Embeddings.

There are several distinct approaches to this problem but all of them are based on the same assumption that the meaning of the word is highly related to its context - the other words that surround it. That leads to a very good property: if two words are close by meaning and, in the most extreme cases, interchangeable in the sentence - they will be very close in vector space and thus their cosine similarity will be close to 1. Such word context can be expressed in a form of cooccurrence matrix. This matrix will be highly sparse and it is a natural solution to apply matrix factorization methods (like SVD) to it, which will result in the matrix of much lower dimensionality and it could be used as the representation. However, with rising popularity of

neural network and their outstanding ability to generate embeddings, more computationally efficient models were found.

The model that has expanded understanding of similarity to the relationship between words, by introducing multiple degrees of similarity with a became-classical example:

$$\begin{aligned} \text{Vec}(\text{King}) - \text{Vec}(\text{Woman}) + \text{Vec}(\text{Man}) &= \text{Vec}(\text{Queen}) \\ \text{Vec}(\text{Paris}) - \text{Vec}(\text{France}) + \text{Vec}(\text{Italy}) &= \text{Vec}(\text{Rome}) \end{aligned}$$

was word2vec by Mikolov et al., 2013. It was achieved due to its outstanding computation performance and ability to process input corpus with over 1B words. Two fully-connected neural network architectures were proposed. The first is Continuous Bag-of-Words (CBOW) model. It accepts as input one-hot encoded vectors of surrounding words, pass them through fully-connected projection (hidden) layer and tries to predict the middle word (classification task) on output fully-connected layer with softmax. The second, in opposite, having middle word as input tries to maximize the classification of surrounding words. In both cases, projection layer can be used as embedding lookup matrix after training. Weight matrix for this layer will have dimension  $N \times D$  where  $N$  - is the amount of words in vocabulary and  $D$  - is the dimension of embedding.

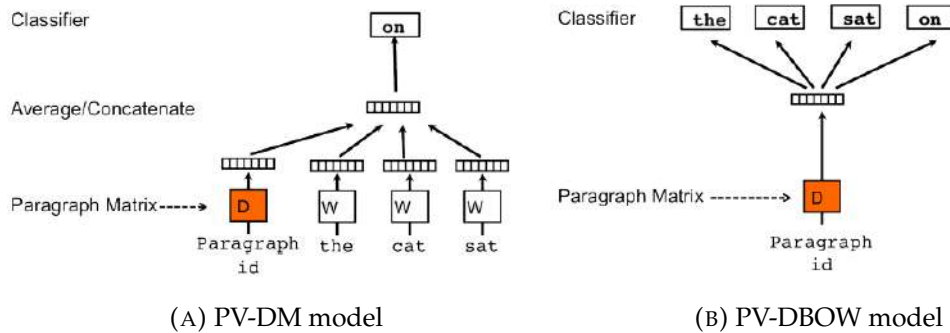


FIGURE 2.4: Architecture of doc2vec models: (A) Paragraph Vector Distributed Memory (successor of CBOW) (B) Paragraph Vector Distributed Bag-of-Words (successor of skip-gram) [Le and Mikolov, 2014]

## doc2vec

The successor of ideas proposed in word2vec became model doc2vec by Le and Mikolov, 2014. In the paper "Distributed Representations of Sentences and Documents" they proposed to use the same CBOW and skip-gram models but with additional paragraph vector as input. That vector is supposed to bring missing context of the full document to the process of word classification. This vector is shared across all word inputs from the same document (or Wikipedia article in our case). Thus, set of this paragraph vectors for all documents in dataset creates the document embedding matrix that can be trained along with word embeddings matrix on the same classification task. In the case of the skip-gram model it is even proposed to omit word vector and leave classification task of context words based on paragraph

vector only. This is called Paragraph Vector Distributed Bag-of-Words model (PV-DBOW). It was reported that the combination of both models showed the best result across a big variety of tasks.

### 2.2.2 Structural representations

Having the approach to construct base (textual) representations we can now move further to the more complex datasets. Graphs - are powerful structures with great expressive power and ability to model complex relationship structures between entities. Graphs can be used as denotation of a large number of systems across various areas including social science (social networks), natural science (physical systems, and protein-protein interaction networks), knowledge graphs and many other research areas [Zhou et al., 2018]. The task of finding embeddings for graph nodes received significant attention in the latest years due to ability to reformulate many Machine Learning problems like classification, clusterization, link-prediction or visualization via simpler models operated with low-dimensional representations [Goyal and Ferrara, 2018].

#### Matrix factorization

Until recent years the most prevalent approach for task of finding representations for graph nodes was matrix factorization. Graph must be expressed in a form of adjacency matrix  $S$  which consists of weights:  $s_{ij}$  = edge weight between  $v_i$  and  $v_j$  if vertices  $v_i$  and  $v_j$  are connected. All other positions in matrix are filled with 0. For unweighted graph, matrix is just being filled with 1 in position  $i, j$  (and  $j, i$  if graph is undirected) when there is connection between corresponding vertices.

By factorization (dimensionality reduction) of such matrix we can obtain the embeddings. However, complexity and required computational resources of this methods is enormous. Time complexity for most algorithms is  $O(|E|d^2)$  according to Goyal and Ferrara, 2018 where  $E$  - is set of all edges of the graph,  $|E|$  - total amount of edges and  $d$  - number of dimensions of resulting embedding. That is unacceptable for real-world graphs like Wikipedia where amount of edges increase quadratically in relation to increasing amount of vertices. On this moment graph of English Wikipedia consists of more than 1B edges.

#### Random walk family

One of approaches to tackle the complexity of Matrix Factorization is to use Random walk algorithms. This approximation algorithms were especially successful in similar tasks, where factorization used to be the leading solution, for example, PageRank [Bahmani, Chowdhury, and Goel, 2010]. In this approach representations are being learned from random walks - paths through graph that consist of randomly chosen steps. That helps to incorporate knowledge about communities among neighborhoods and in the same time requires to observe only some sample of the graph (for one walker), which enables possibilities for parallelization. Time Complexity of the most popular random walk algorithms DeepWalk and Node2vec is  $O(|V|d)$  according to [Goyal and Ferrara, 2018] where  $|V|$  is amount of vertices.

**Node2Vec** is expanding ideas behind skip-gram model - to predict missing word based on its context (or vice versa), but using nodes instead of words and getting context by series of Random Walks. Its objective is to maximize the log-probability of observing a network neighborhood  $N_{S(u)}$  for a node  $u$  conditioned on its feature representation, given by  $f$  [Grover and Leskovec, 2016]:

$$\max_f \sum_{u \in V} \log \Pr(N_{S(u)} | f(u))$$

This objective is being optimized with Stochastic Gradient Descent. Original implementation, however, is poorly scalable and demands huge memory resources [Zhou, Niu, and Chen, 2018].

### 2.2.3 Deep Representation Learning

We discovered embedding producing techniques that preserve semantic properties or structure properties. But to get the best representation we must utilize both of them in the same model. This idea has found its application in Neural Network-based models.

Wang, Cui, and Zhu, 2016 proposed to use deep autoencoder which takes each node's neighborhood as input and tries to reconstruct its features after encoding and decoding. But this model has two limitations that make it impossible to use it with massive real-world graph. First, it requires all neighbors to be passed as input - full adjacency matrix, which in case of highly sparse graph leads to high memory consumption ( $O(|V|^2)$ ). Second, which is derived from the first, it learns weights for encoder and decoder with static predefined shape, based on  $|V|$  - which means, those weights cannot be used if we add just one more node to the graph. In addition such big amount of neurons can be a reason to very long training and possibly not converging at all. Overall, this model showed good performance on cleaned small-sized datasets but is not a very good fit on practice [Grover and Leskovec, 2016].

More scalable approach was introduced by Kipf and Welling, 2016 - Graph Convolution Network (GCN). Convolution Neural Networks showed state-of-the-art performance in many Computer Vision tasks and especially representation learning for high-dimensional data like images, videos, sound. Thus, it is not a surprising that there have been a lot of research recently with aim to generalize NN to arbitrarily structured objects like graphs.

GCN is a multi-layer network, where each layer can be formulated as:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l+1)}) \quad (2.10)$$

where  $\tilde{A} = A + I$  - is adjacency matrix with self-connections ( $I$ ),  $\tilde{D} = \sum_j \tilde{A}_{ij}$ ,  $W$  - trainable weights and  $H$  is output of previous layer or  $H^{(0)} = X$  is input,  $X$  - is node features. If we will put that in simple words: each node collects features of its neighbors that were propagated through trainable filters (convolutions) - so called, message passing. On each step (layer) node collects knowledge of its neighborhood and propagates its state further on the next step. Thus, properties of 1st, 2nd, ..., nth proximity are being incorporated into node's state along with preserving original features of node's community. But, this model has similar limitations as described

autoencoder - it works with full adjacency matrix, which makes it harder to work with big graphs due to memory boundaries and, in addition, restricts adding new nodes into the graph without full retraining of the model.

## GraphSAGE

To tackle this limitations GraphSAGE model was introduced by [Hamilton, Ying, and Leskovec, 2017a]. On one hand it is based on well-grounded GCN [Kipf and Welling, 2016] which is - currently - state-of-the-art for Deep Learning Graph embeddings approaches. On the other hand, it arises from poorly practical tasks and solves GCN inability to scale. This model improves GCN in a way that only some fixed-sized sample of neighbors is utilized on the Convolutional Layer. All states (initialized with node features) from sampled neighbors are being aggregated and assigned as node's current state. This process repeats and state is being propagated further. There are several proposed aggregators - from simple mean of vectors, element-wise maxpool or meanpool or LSTM cell. Because of fixed-size samples we also have fixed-sized weights that are generalized and could be applied to new unknown part of the graph or even completely different graph. Thus, with inductive learning, we can train the model on a sub-graph, which means less computation resources are required, and evaluate generalization on the full graph.

Let us rewrite (2.10) with GraphSAGE changes:

$$h_{NS(v)}^{(l+1)} = AGGREGATE_k(\{h_u^{(l)}, \forall u \in NS(v)\}) \quad (2.11)$$

$$h_v^{(l+1)} = \sigma(W_a^{(l+1)} CONCAT(h_v^{(l)}, h_{NS(v)}^{(l+1)})) \quad (2.12)$$

where  $NS(v)$  - is neighborhood sample of node  $v$ , which, as we mentioned before, has constant size. Equations (2.11-12) are approximation of layer function in GCN - non-linearities were kept in place, but due to sampling we replaced linear projection of the whole neighborhood  $W^{(l+1)}$  with aggregation step and simpler linear mapping  $W_a^{(l+1)}$  along with preserving previous state of the node  $h_v^{(l)}$ . Time complexity of the model is fully dependent on chosen sample sizes  $O(\prod_{l=1}^K S_l)$ , where  $S_l$  - is size of sample on  $l$  layer.

There are two distinctive ways to learn weights  $W_a^{(l)}$ : supervised and semi-supervised.

In semi-supervised setup model is trained only from self-sufficient knowledge about graph structure and can utilize given node features or generate "identity" features if there is none (which will disable prediction for unknown nodes). With xent-like loss function (2.13) we encourage connected or close in the graph nodes to have similar representations and in contrast punish the model for placing all nodes too close in space with help of negative sampling (the same approach as in doc2vec).

$$J(z_u) = -\log(\sigma(z_u^T z_v)) - QE_{v_n \sim P_n(v)} \log(\sigma(z_u^T z_{v_n})) \quad (2.13)$$

where  $u$  and  $v$  are close-lying nodes with  $shortest\_distance(u, v) < N$  where  $N$  - is random walk max length;  $P_n$  - is negative sampling distribution;  $Q$  defines the number of negative sample.

However, it was shown by Ying et al., 2018 and confirmed by our experiments that this loss function is not always effective. They proposed to replace it with max-margin loss for every pair of nodes in learning context:

$$J(z_u z_i) = E_{v_n \sim P_n(u)} \max\{0, z_u \cdot z_{v_n} - z_u \cdot z_i + \Delta\} \quad (2.14)$$

where  $\Delta$  denotes the margin hyper-parameter.

Alternatively, any supervised task that can produce useful representation, for example, node classification can be used. In case of classification task softmax multiclass log-loss can be applied.

### Aggregator functions for GraphSAGE

Aggregator function operates over an unordered set of vectors sampled from node neighborhood, thus this function must be invariant to possible permutations of input data [Hamilton, Ying, and Leskovec, 2017a].

**Mean aggregator** is the simplest possible solution which guarantees fast forward and backward propagations with preserving decent level of neighborhood representation. We take elementwise-mean across sampled vectors  $NS(v)$ :

$$AGGREGATE_{mean} = \text{mean}(\{h_u^{(l)}, \forall u \in NS(v)\}) \quad (2.15)$$

**Meanpooling aggregator.** In this approach additional fully-connected layer with activation is being applied to each input vector before aggregation. We can consider this as projection  $R^u \rightarrow R^f$ , when  $R^f$  is usually of significantly higher dimension than  $R^u$ . Another words, this layer captures different aspects of the neighborhood set [Hamilton, Ying, and Leskovec, 2017a] by extracting additional features:

$$AGGREGATE_{meanpool} = \text{mean}(\{\sigma(W_{pool} h_u^{(l)} + b), \forall u \in NS(v)\}) \quad (2.16)$$

**Maxpool aggregator.** Similar to previous approach but with different reducing function - element-wise max - is also worth to test in our experiments:

$$AGGREGATE_{maxpool} = \max(\{\sigma(W_{pool} h_u^{(l)} + b), \forall u \in NS(v)\}) \quad (2.17)$$



## Chapter 3

# Recommending Articles to Wikipedia Editors

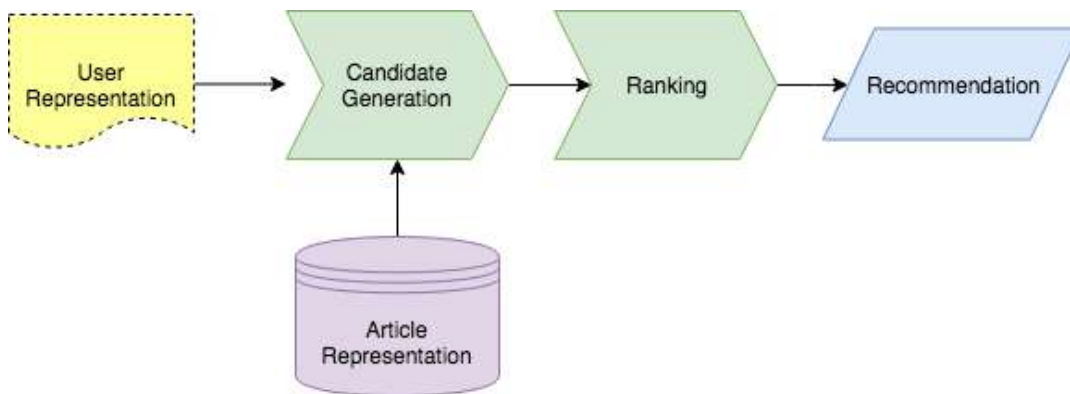


FIGURE 3.1: General flow of article recommending system

We propose a solution for the task of recommending the next article to edit to Wikipedia contributors combined from different parts described in previous Chapter 2. General design of our solution (Figure 3.1) is inspired by classic *Information Retrieval* architecture. First we represent users by the articles that they have edited, then we generate a list of candidates from the article pool, based on a given user by comparing that user and the article representations. Next, we sort our article candidates accordingly to the user preferences and generate a list of *top-n best candidates* recommendation.

The primary challenge for our system is producing good user and article representations. This is a crucial part for candidate generation (Figure 3.2). It is an especially big problem for user representation since most of Wikipedia contributors do not fill any additional information about themselves except their login, and around 28% of all revisions in our English Wikipedia dataset, are being done by anonymous users. The only useful information that could uniquely characterize the user is the history of his editions. Hence, most of our efforts were dedicated to learning articles' representations.

One of the effective approaches to construct good representations is to learn them with recommendation supervision as it was done for YouTube and Pinterest recommendation systems [Covington, Adams, and Sargin, 2016; Ying et al., 2018]. However, it is not possible to follow this approach due to the lack of the required comprehensive-enough dataset of previous interactions. History of users editions

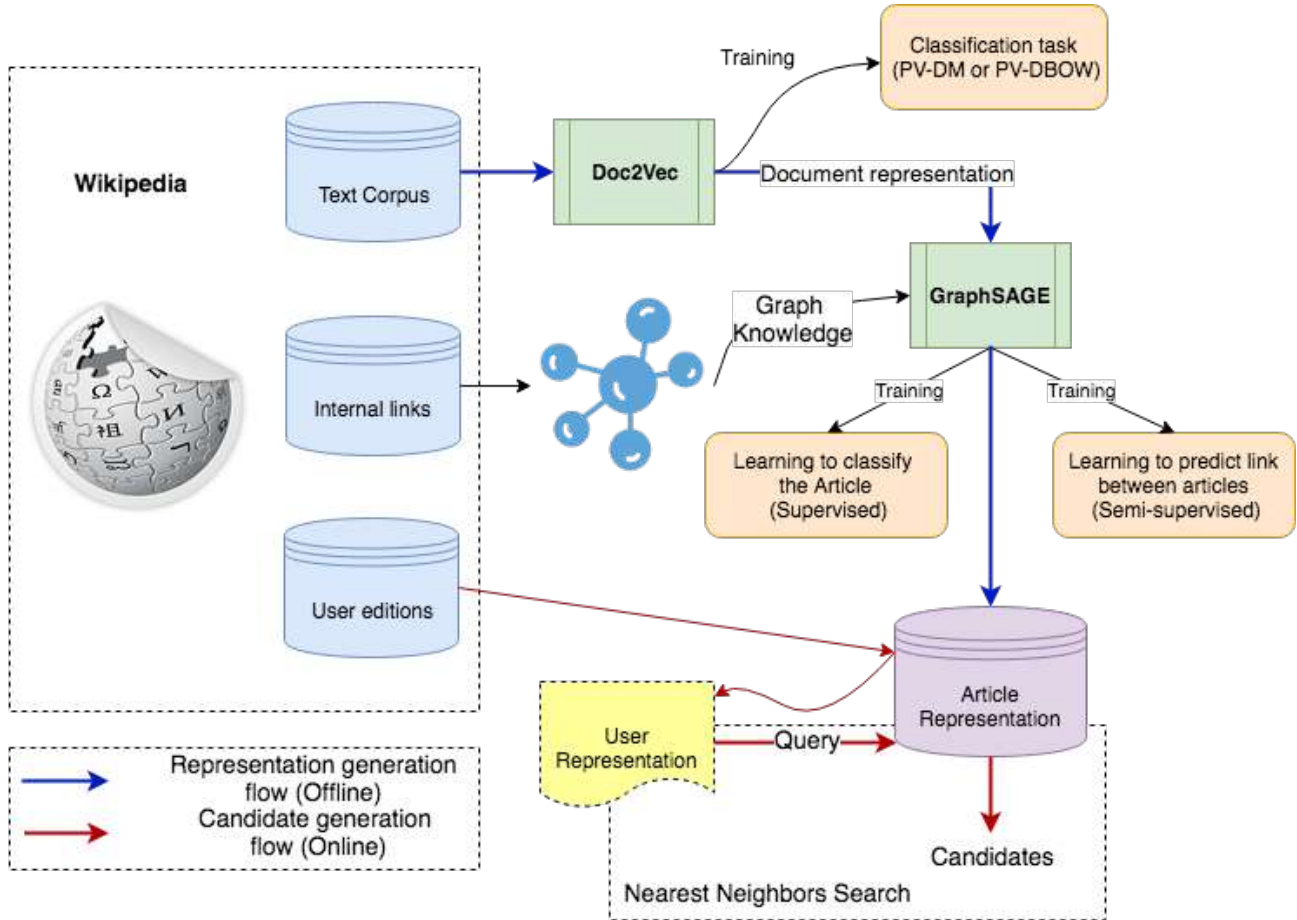


FIGURE 3.2: Flow of Candidate Generation for Wikipedia articles recommendation: Doc2Vec embeddings are trained on Wiki Text Corpus and then passed as input into GraphSAGE model which is being separately trained in two different setups with graph of internal links between articles. Received articles' representations are then used in Nearest Neighbors Search to produce candidates

in Wikipedia is far from exhaustive (88% of users of English Wikipedia done less than 5 major editions and 94% - less than 10) and too sparse in a way that it is hard to model user's area of interest (see section 4.1.3). Therefore, the additional challenge is to train Representation Learning in semi-supervised (or even unsupervised in relation to our final task) way. **One of the main contributions of our work is the examination of alignment of unsupervised constructed representations to the task of finding best recommendations.**

For generating candidates we first calculate representation vectors for all articles in our dataset. Then for the given user we define his representation as element-wise average of vectors of corresponding articles that were edited by this user. Next, we conduct Nearest Neighbors search with user representation as *query* in the articles' representation database.

For learning best article representation text features are needed to be extracted first. We train Doc2Vec model on Text Corpus of all Wikipedia articles in a given language. Output vectors of Doc2Vec are being passed as input features to the Graph Convolutional Network. GraphSAGE is being used as GCN due to ability learn with an inductive approach and construct embeddings for unseen nodes. During



preprocessing of input dataset - snapshot of Wikipedia Database - we create a graph  $G(V, E)$  where  $V$  denotes set of articles, and  $E$  - set of links between them. GraphSAGE is utilizing structure knowledge from graph  $G$  and produces new vectors that preserve both text and structural representations.

We train GraphSAGE model with two different setups. In the first case model's goal is to predict classes of a given article (multi-class multi-label problem) with cross-entropy loss function. The other setup does not utilize any manually created label, and is purely based on the graph structure. Model is being learned to predict links between given articles (denoted as  $E$ ) with max-margin loss function (Eq. 2.14). We will separately evaluate resulting representations received from both of these setups.

Due to the inductive nature of GraphSAGE architecture, we do not need to re-train the model every time after adding a new article into the database. After producing document vector (which requires some minor retrain of Doc2Vec) and updating Graph  $G$  structure - we can run GraphSAGE model as is, with already trained weights.

In serving time, recommendation candidates will be produced by applying k-Nearest-Neighbors search algorithm to find the most similar articles to user representation vector in the pre-computed database of all articles' representations.

**In the second part of our system,** candidate ranking, we are trying to model user preferences based on previous edit history of Wikipedia contributors. With given previous editions and articles, we produce a relevant a list of candidates, ranked by its relevance for a given user. Our model is trained on binary labels - *relevant* / *not relevant* (logistic regression) but on serving time it will produce probabilities of user interest, which could be used as a sorting key.

This approach is inspired by Point-wise ranking (described in Chapter 2) and is implemented in many similar RS: YouTube, Google Play, eBay. The model is shown on Figure 3.3 and consists of several fully-connected layers with ReLU activation except for the last layer, where sigmoid activation is used. As input model accept a concatenated vector of user and candidate representations.

Pretrained Deep-Ranking model in serving time will sort received on the first stage candidates by relevance probability and first K articles in sorted list (with highest probability) could be returned to user as recommendations.

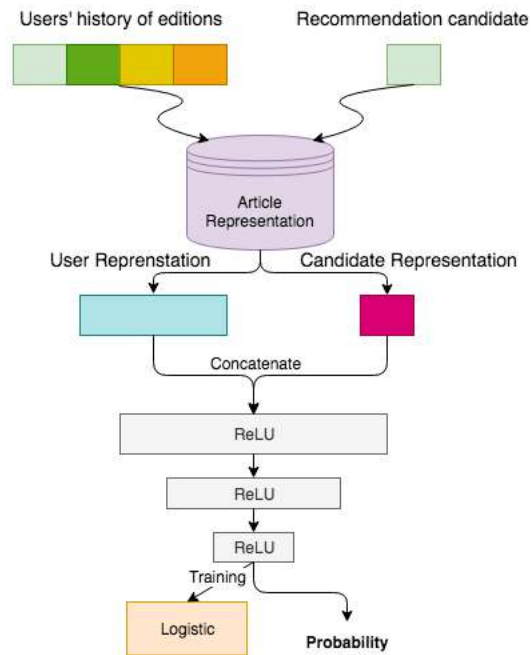


FIGURE 3.3: Candidate Ranking: user previous history of edited articles along with candidate are being passed through articles' representation database (Embedding Layer) and then through several fully-connected layers to train in the log-regression setup

## Chapter 4

# Experiments and Evaluation

### 4.1 Introducing Wikipedia dataset

page	pagelinks	redirect
page_id INT	pl_from INT	rd_from INT
page_namespace INT	pl_from_namespace INT	rd_namespace INT
page_title VARCHAR(255)	pl_namespace INT	rd_title VARCHAR(255)
page_restrictions TINYBLOB	pl_title VARCHAR(255)	rd_interwiki VARCHAR(32)
page_is_redirect TINYINT		rd_fragment VARCHAR(255)
page_is_new TINYINT		
page_random DOUBLE		
page_touched BINARY(14)		
page_links_updated VARBINARY(14)		
page_latest INT		
page_len INT		
page_content_model VARBINARY(32)		
page_lang VARBINARY(35)		

FIGURE 4.1: Structure of tables from Wikipedia Database imported from SQL dump to build the Article Graph <sup>1</sup>

Let us now introduce the dataset that we used for training representation and ranking models and also for offline evaluation of our Recommender System. Our system was built and evaluated on data collected from Wikipedia - a multilingual, web-based, free encyclopedia<sup>2</sup>. There are many independent **editions** of Wikipedia - usually one per each language (English Wikipedia, German Wikipedia, Spanish Wikipedia, etc). We tested different parts of our system against different editions, but for the final end-to-end evaluation Ukrainian and English editions were selected.

All data used in our project has been downloaded from official Wikimedia Dump Storage<sup>3</sup>. There is separate database, and hence, separate list of dumps for each Wikipedia edition. Wikipedia dump - is a snapshot of its database, that is being generated once or twice a month and consists of full Wikipedia state on the moment of generation: text from all articles, links to media, article categories, links between articles, article revisions and comments for them, etc. Some of objects (like pages or categories) are being stored in SQL format, others, that implies more deep structure (like articles with revisions, information about users that created new revision and their comments) are stored as XML dumps.

<sup>1</sup>Source: [https://www.mediawiki.org/wiki/Manual:Database\\_layout](https://www.mediawiki.org/wiki/Manual:Database_layout)

<sup>2</sup><https://en.wikipedia.org/wiki/Wikipedia>

<sup>3</sup><https://dumps.wikimedia.org/>

TABLE 4.1: Specifications of built Wikipedia Graphs

Specification	Source	
	English Wikipedia	Ukrainian Wikipedia
Amount of vertices ( $ V $ )	5251875	770650
Amount of Edges ( $ E $ )	458867626	62245214
Average Degree ( $\overline{d_{all}}$ )	174	160
Median Degree ( $d_{all}$ )	60	60
Approx. Diameter (D)	23	119
Amount of labeled nodes	4652604	294825


First of all, for representation learning we built a graph  $G(V, E)$ , where set of nodes  $V$  is the set of all Wikipedia pages belonging to article namespace<sup>4</sup> and  $E$  is set of directed links between them. SQL dumps of 'page', 'pagelinks', 'redirects' tables (Figure 4.1) were parsed to organize this data.

During pre-processing stage, all links to redirect pages<sup>5</sup> were replaced by their actual destinations. "Category pages", that consists only of links to other pages and do not have their own content, were detected and filtered out. They can be detected by very high rate outgoing links/incoming links.

We used Apache Spark<sup>6</sup> for parallel parsing of SQL dumps and Spark GraphX<sup>7</sup> for discovering and cleaning Article Graph. Received Graph was converted into binary format with graph-tool<sup>8</sup> to achieve fast loading and processing in model training. Some useful characteristics of resulting graphs (built for editions used in evaluation) can be found in Table 4.1.

#### 4.1.1 Labels for Supervised learning

For training GraphSAGE in supervised fashion with classification cross-entropy loss function labels were required. Several semi-supervised approaches were tested to collect labels. In particular we applied two community detection algorithms to our graphs: Weighted Cluster Coefficient (WCC) (non-overlapping) [Prat-Pérez, Dominguez-Sal, and Larriba-Pey, 2014] and BigClam (overlapping) [Yang and Leskovec, 2013]. In our experiments with labeling English Wikipedia WCC detected 70K cluster, which were not stable for each different run - results simply did not converge. In tests with BigClam (we used implementation provided by SNAP<sup>9</sup>) on machine with 4 CPU and 26Gb of memory it took on average 52 hours to run one iteration for each *count-of-communities* guess since we ran it with autodetection of amount of communities.



categorylinks	
cl_from	INT
cl_to	VARCHAR(255)
cl_sortkey	VARBINARY(230)
cl_sortkey_prefix	VARCHAR(255)
cl_timestamp	TIMESTAMP
cl_collation	VARBINARY(32)
cl_type	ENUM(...)

FIGURE 4.2: Structure of Categorylinks Table used for building labels in supervised training

<sup>4</sup><https://en.wikipedia.org/wiki/Wikipedia:Namespace>

<sup>5</sup><https://en.wikipedia.org/wiki/Wikipedia:Redirect>

<sup>6</sup><https://spark.apache.org/>

<sup>7</sup><https://spark.apache.org/graphx/>

<sup>8</sup><https://graph-tool.skewed.de/>

<sup>9</sup><http://snap.stanford.edu/>

We found more appropriate to utilize hand-crafted labels from Wikipedia crowd-sourced initiative on categorizing all articles<sup>10</sup>. Those categories were imported from dump of ‘categorylinks’ table (Figure 4.2) and, hence, we ended up with multi-class multi-label learning task with 1100 overlapping classes.

### 4.1.2 Text for document representation

For extracting articles’ texts we took latest revision per each article from XML dump of all revisions (structure given on Figure 4.3). **Article’s revision** - is a specific version of article’s content that is created after each new **modifications** done by user. Wikipedia article’s content consists of a lot of system information: from links to other pages, links to media files, fact references to math content, tables and markups. We utilized Gensim library<sup>11</sup> to extract only useful information, tokenize and lemmatize text and prepare for model (doc2vec) training. We tagged each document with article id (page id) to be able to look up the vector of each specific article in output matrix produced by the model.



revision	
rev_id	INT
rev_page	INT
rev_text_id	INT
rev_comment	VARBINARY(767)
rev_user	INT
rev_user_text	VARCHAR(255)
rev_timestamp	BINARY(14)
rev_minor_edit	TINYINT
rev_deleted	TINYINT
rev_len	INT
rev_parent_id	INT
rev_sha1	VARBINARY(32)
rev_content_model	VARBINARY(32)
rev_content_format	VARBINARY(64)

FIGURE 4.3: Structure of Article’s Revision table imported from XML dump for Doc2Vec training

### 4.1.3 Revision history for evaluation

For evaluation of our recommendation system in end-to-end fashion we utilized data about articles’ revisions (Figure 4.3) and reorganized it into *revisions-per-user* dataset. Only revisions that were created after January 1, 2015 were kept in this dataset, so our recommendations that are based on the latest snapshot of article graph (October 2018) will not recommend too many articles that did not exist at the moment of modification. Some interesting statistics about this dataset is shown on Figure 4.4. It is easy to notice that huge part (88% for English edition) of contributors are not regular users - they edited less than 5 different articles for selected dates. Thus, it is very hard to model their behaviour and their data is unsuitable for our evaluation. The part of contributors that fits to our needs has mostly edited from 5 to 40 different articles, though diversity (calculated accordingly to Eq. 2.9) of those articles is rather high (Figure 4.4 C, D). That is the main cause our representations cannot be trained against this data like it was done in YouTube recommendation system [Covington, Adams, and Sargin, 2016] - training dataset is small (around 60K of users from English Wikipedia and less than 7K users from Ukrainian) and users’ area of interest is too sparse.

<sup>10</sup>[https://en.wikipedia.org/wiki/Wikipedia:WikiProject\\_Categories](https://en.wikipedia.org/wiki/Wikipedia:WikiProject_Categories)

<sup>11</sup><https://radimrehurek.com/gensim/>

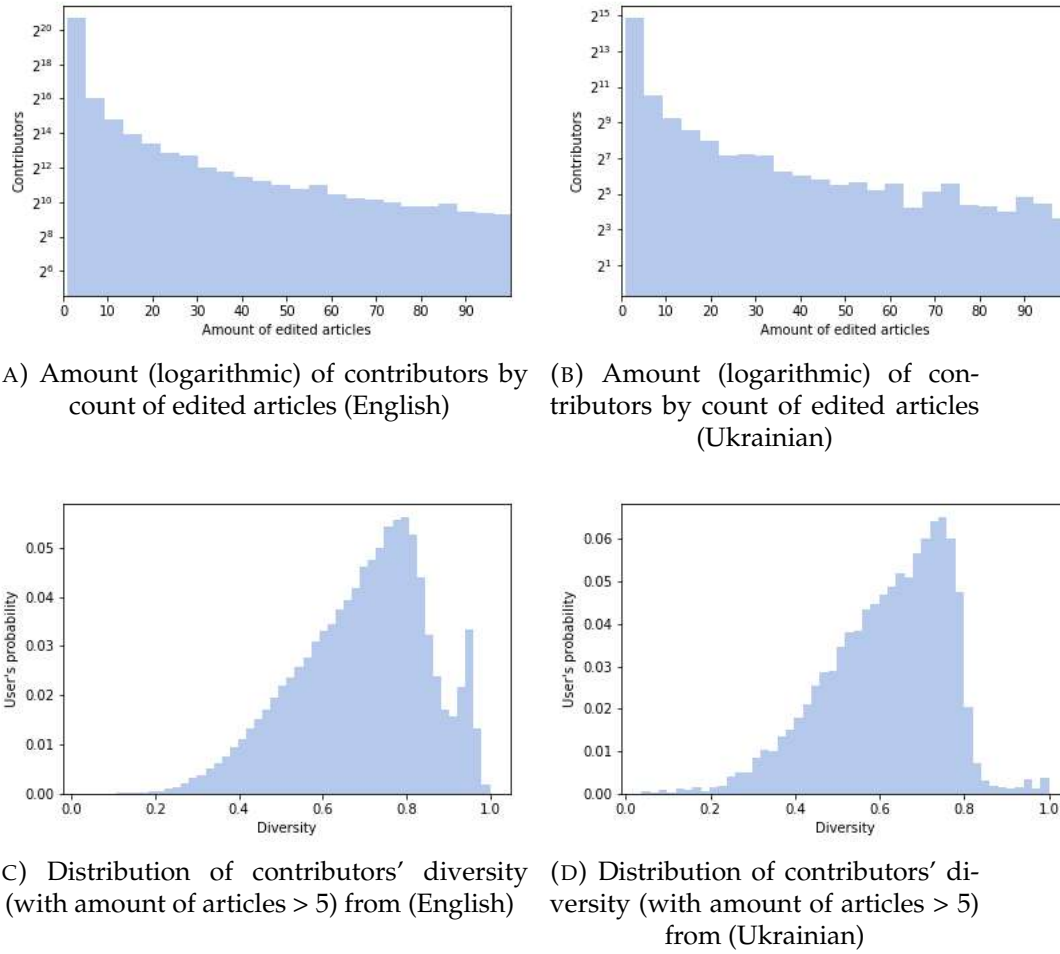


FIGURE 4.4: Researching Wikipedia's contributors: distribution by amount of edited articles (A) and editions' diversities (B)

## 4.2 Efficient (Big) Data Processing

### 4.2.1 Parallel Wikipedia Dumps pre-processing

Our main challenge in building the system was the amount of data that has to be processed before starting to train our models: to clean up and extract only useful part from the Wikipedia Database. We consider it as typical Big-Data problem. As already was mentioned we worked with SQL (MySQL format) and XML dumps. Some of this dumps were especially huge. For example, full XML dump of revisions for the English Wikipedia, with included exact text of every change, in uncompressed state has size more than 600Gb. We wanted to process this data in parallel. On the other hand all experiments must have been conducted with different Wikipedia editions, so we needed to introduce automatization into this preprocessing as well.

Another example is SQL dumps. Toolset provided by MySQL Database includes *mysqlimport* util. However, it only works in sequential order to follow all transaction restrictions. That became an issue when we were importing pagelinks. There

are more than 1B rows in original *pagelinks* table in English edition. We experimented first with smaller datasets. The import process of *pagelinks* from Ukrainian Wikipedia took around 26 hours due to bottleneck in disk operations (on machine with regular HDD) since `mysqlimport` wraps each query into separate transaction. English Wikipedia is approximately 8 times bigger and since importing is sequential we assume it would take 8 times more time. There are some solutions trying to solve transaction issue, but there are no solutions to our knowledge enabling parallel load. We wanted to set up different Wikipedia editions and try our models in different environments - so there was a need for parallel read solution for MySQL format as well as for XML.

To tackle this issues we developed our own preprocessing toolkit that can be also useful in future researches that would involve Wikipedia dataset<sup>12</sup>. It is based on Apache Spark and includes parallel import of SQL-dumps, fast data-mining and some Graph cleaning routines. With this toolkit we were able to decrease time spent on preparation - for all next editions it took less than 8 hours using cluster of 8 machines (4 CPU, 26 Gb Memory) to get all data cleaned and ready for training (tested on English, Spanish, Russian, Ukrainian editions).

#### 4.2.2 GraphSAGE Scalable Implementation

The second part of our implementations contribution is related to the code-base of the GraphSAGE model. At first we started our experiments with original code-base<sup>13</sup> provided by authors of the paper [Hamilton, Ying, and Leskovec, 2017a]. However, despite the fact that model is designed to work with massive graphs with over 1B edges, in our experimental setup the code crashed with 100M of edges. First of all, their graph processing is built on library `networkx`<sup>14</sup> written completely in Python. This library has good API and reach functionality but was created for research-only purposes to work with educational lab graphs. It expects input graph to be stored in json (text) format and stores all nodes and edges as separate python objects which leads to very high memory consumption - according to our tests it could take up to 1kB per edge, so it would require just over 400GB of memory just to load graph of English articles. We reworked all data consumption model, rewrote batch feeding processing, variables initialization and propagation in the model. Resulted code was published on github<sup>15</sup>.

### 4.3 Experiments

#### 4.3.1 Document Representations

We trained both Doc2Vec-DM and Doc2Vec-DBOW models on Wikipedia Corpus with vector size = 100 and window size = 6. All words that occurred less than 20 times in all texts were filtered out. Results have been visualized on Figure 4.5 as distribution of distances between vectors of random articles. That represents how good

<sup>12</sup><https://github.com/pyalex/wiki2graph>

<sup>13</sup><https://github.com/williamleif/GraphSAGE>

<sup>14</sup><https://networkx.github.io/>

<sup>15</sup><https://github.com/pyalex/GraphSAGE>



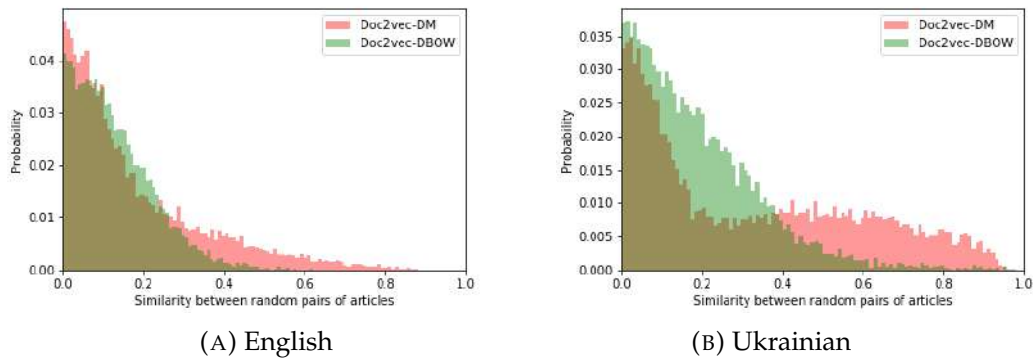


FIGURE 4.5: Comparison of received representation by training Doc2Vec with different settings: Distributed Bag-of-Words (DBOW) and Distributed Memory (DM). Figure shows distribution of similarities between two articles randomly selected from Database

vectors are distributed in the space. Vectors generated by Doc2Vec-DBOW were chosen as input to the GraphSAGE model since this model showed better performance on the offline evaluation (Tables 4.3, 4.4)

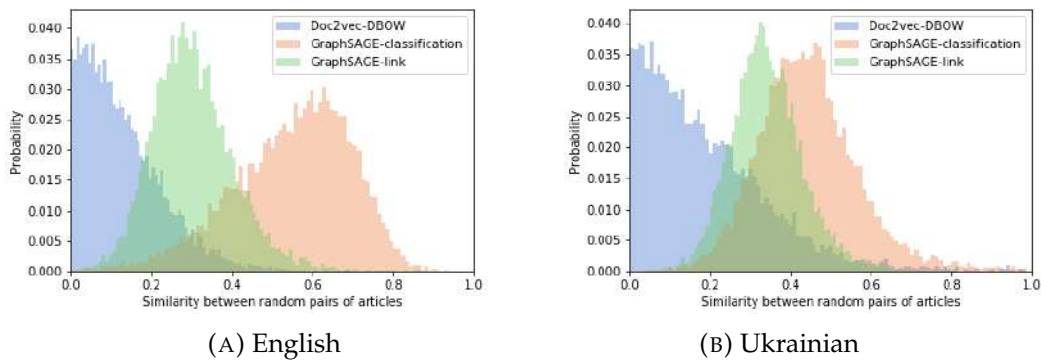


FIGURE 4.6: Comparison of received representation from Doc2Vec, GraphSAGE-classification, GraphSAGE-link-prediction. For GraphSAGE model with *meanpool* aggregator has been used. Figure shows distribution of similarities between two articles randomly selected from Database

### 4.3.2 GraphSAGE

As was mentioned above GraphSAGE became our major option for representation construction based on both structural and semantic data.

**Training details.** For all training experiments with GraphSAGE we generated sampled adjacency matrix based on Articles' graph  $G$ . This matrix was constructed once and used across all trainings for one Wikipedia edition. That allowed us to speed-up our experiments and guarantee comparability of different setups. Width of adjacency matrix was determined by our experiments, available memory resources and graph statistics (Table 4.1). We selected 128 as maximum amount of neighbors in this matrix. If node had more than that - random subsample was selected. On each

convolutional step we picked random sample of 25 neighbors from this adjacency matrix. This 25-neighbors sample is being resampled on each new batch. For better generalization we used batch size 512, since experiments with Dropout [Srivastava et al., 2014] between convolutional layers led to no improvement in generalization. We focused on three aggregators: mean, meanpool, maxpool (Eq. 2.15-17) and set-up size for all output vectors to 256 as balance between better resolution and available memory.

**Link Prediction.** At first, experiments with self-sufficient training were conducted. Document representations (from doc2vec) were passed as initial node states and graph edges played the role of labels when the model was trying to predict those edges - links between nodes. Xent-like loss (Eq. 2.13) were applied to the learning task. However, in this setting model did not converge with any of aggregator function. We manually tested generated representations by searching neighbors with cosine similarity and our results were no better than random. We conduct manual review for 100 randomly chosen articles from database.

After we switched to max-margin loss (Eq. 2.14) that was recommended by Pinterest paper [Ying et al., 2018], we observed improvement. This results are referenced as *GraphSAGE-link-prediction* in our evaluation.

**Training with Classification and labels' issues.** Alternatively, we trained Graph Convolutions on supervised classification task. For this purpose hand-crafted categories from WikiProject<sup>16</sup> were utilized. This approach has a limitation. Despite the fact, that almost 90% of articles in English edition have category WikiProject covers *only* articles in English<sup>17</sup>. As solution for this problem - we utilized **WikiData**<sup>18</sup> database which connects articles across different editions to propagate categories to other languages. However, Wikidata is not 100% reliable: first, it is not always one-to-one relation between pages in different editions, and wikidata is not designed to handle such situations; and second, there could be many articles in, let us say, Ukrainian Wikipedia that are absent in English, thus, again, there is no category to propagate. As the result, there are only about 38% of articles that have categories in Ukrainian Wikipedia after we mapped english-ukrainian articles with WikiData and propagated labels (Table 4.1).

The loss function for this setup can be formulated as:

$$J = - \sum_{c=1}^C target_c \log(\text{sigmoid}(\text{prediction}_c)) \quad (4.1)$$

where C - is total number of classes.

Comparison of received representations from Doc2Vec, GraphSAGE with link prediction and GraphSAGE with classification task is shown on Figure 4.6. Model trained with link-prediction preserves higher sparsity in the vector space especially in English edition, where embeddings generated by GraphSAGE-classification are positioned much closer and, as we will see in final results (Tables 4.3, 4.4), that is

<sup>16</sup>We use Wikiprojects as user-generated tags, such as "History", "Sport" or other more specific such as History of France", for more details on Wikiprojects please refer to: [https://en.wikipedia.org/wiki/Wikipedia:WikiProject\\_Categories](https://en.wikipedia.org/wiki/Wikipedia:WikiProject_Categories)

<sup>17</sup>There are WikiProjects in other languages, however currently their coverage is limited

<sup>18</sup>Wikidata is manually generated knowledge base. For details please go to: [https://www.wikidata.org/wiki/Wikidata:Main\\_Page](https://www.wikidata.org/wiki/Wikidata:Main_Page)



correlating with lower performance (similar to Doc2Vec-DM results).

### 4.3.3 Deep Ranking

For training Ranking model dataset from users' history of modifications was constructed. As input this model takes 5 articles previously edited by user (as representation of user preferences) and 1 candidate that might interest the user. The model tries to predict the probability of relevance of this candidate to the current user. Those 6 input articles are being passed through *Embedding Layer* populated from representations received from GraphSAGE (Figure 3.3) and then concatenated into one vector. We took positive candidates from actual user history and generated negative candidates with kNN search on constructed articles' representations. Logistic regression with class-weights (due to high class imbalance in real data) was used as loss function on this model's training.

The final architecture of Deep Ranking model consists of 4 fully-connected layers with Batch Normalization [Ioffe and Szegedy, 2015] before each ReLU activation (except for last layer) and Dropout [Srivastava et al., 2014] after each activation (except for last layer). The last layer has sigmoid activation. The level of dropout and amount of neurons on each layer were optimized for each Wikipedia edition separately.

### 4.3.4 Time optimization for kNN search

As it was discussed in section 2.1.1 kNN search is main part of candidate generation and its performance and time and resource consumption is very critical for online recommendation in high-load system. We conduct experiments with different optimizations for kNN: Locality-Sensitive Hashing (LSH), Inverted file with exact post-verification (IVF), Hierarchical Navigable Small World graph exploration (HNSW). Our tests showed that HNSW gives the best speed along with exactly the same recall and MRR as exact search, so with no trade-off in performance we achieved 20x times improvement in speed.

Algorithm	Setup time (s)	Time/request (s)	Recall	MRR
Exact search (inner dot product)	3.91	0.81	0.224	0.0220
IVF	207.02	0.07	0.206	0.0212
HNSW	232.68	0.04	0.224	0.0220
LSH	472.31	0.15	0.215	0.0219

TABLE 4.2: Time Performance of different algorithms for kNN search. All tests were conducted with English articles representation database ( $|V| = 5251875$ ). All implementations were provided by FAISS<sup>19</sup>. Hyperparameters used: IVF (nlist=100), HNSW(size=32 bytes), LSH(bits=256)

<sup>19</sup><https://github.com/facebookresearch/faiss/wiki/Faiss-indexes>

## 4.4 Evaluation

We evaluated our Recommendation System in end-to-end fashion, when system was built accordingly to Figure 3.1.

To prepare evaluation dataset, we subsampled windows with size 10 from user's history of modifications (from users that was not previously used for training or testing Deep Ranking model). First 5 articles denoted user area of interest. To receive user vector we took element-wise average of representations from first 5 articles (GraphSAGE representations). We were trying to predict the rest 5. Algorithm can be expressed as follows:

1. Take first 5 articles. Calculate average of their embeddings vectors, receive user representation as vector
2. Generate candidates by nearest neighbors search of user representation
3. Sort candidates according to ranking algorithm and select first  $K$ . In our evaluation we compare two ranking techniques: sort by cosine similarity; sort by probability from Deep Ranking model
4. Compare Top- $K$  recommendations with the rest 5 articles (from second half of window)

To evaluate the outcome we used two metrics: Recall (Eq. 2.5), that indicates proportion of correctly predicted recommendations from actual history and MRR (Eq. 2.7), that indicates on which positions those correct predictions were in recommendation list. Those metrics are being averaged across all true values.

Results of evaluation are presented in Tables 4.3 and 4.4 for English and Ukrainian Wikipedia editions respectively.

### 4.4.1 Result Interpretation

Let us now consider recommendation example in details to better understand how results shown in Tables 4.3 and 4.4 were received and can be interpreted. Visualization for this example is shown on Figure 4.7, where all representations were decomposed to 2 dimensions using Principal Component Analysis (PCA).

From user's history we selected 5 sequential contributions, obtaining a list of articles per user. For example: [*Petticoat Junction*; *The Lucy Show*; *Pistols'n'Petticoats*; *The Jackie Gleason Show*; *The Tonight Show Starring Johnny Carson*] and her next 5 contributions: [*Bewitched*; *Your Show of Shows*; *Here's Lucy, Karen (1975 TV series)*; *Upstairs, Downstairs (1971 TV series)*]

Based on first 5 revisions we calculated user representation and then Top- $K$  recommendations. We can imagine result of RS as sorted list with each element's position attached to its value:

The Red Skelton Show	...	<b>Here's Lucy</b>	...	<b>Bewitched</b>	Jack Benny	...
1	..	6	...	30	31	...

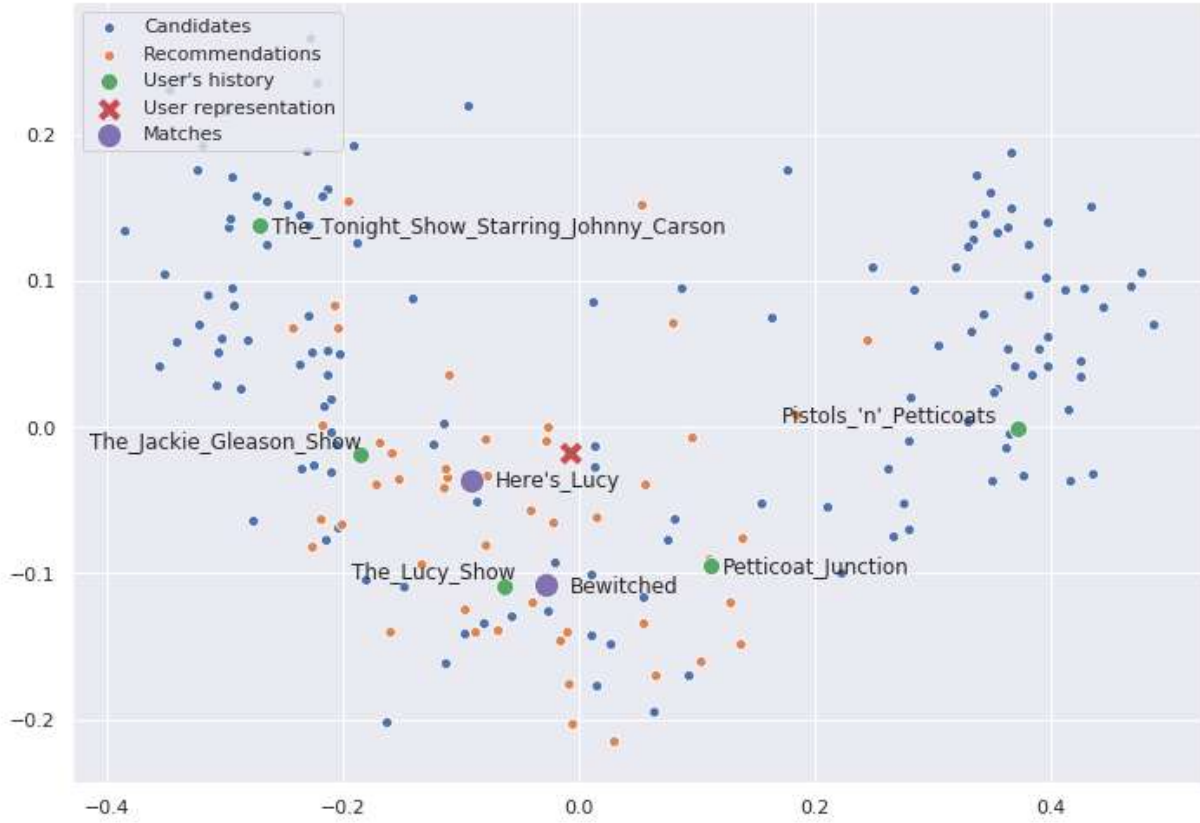


FIGURE 4.7: Vector Representations of user (received by element-wise average from representations of his previously edited articles) and received recommendations on the task of predicting next 5 articles edited by user (PCA decomposed)

From this recommendation list two elements are matched to our true values (as it shown on Figure 4.7). Their positions are 6 and 30. Now we can calculate our evaluation metrics Recall and MRR for this recommendation:

$$\begin{aligned}
 recall &= \frac{t_p}{|T|} = \frac{2}{5} = 0.4 \\
 MRR &= \frac{1}{|T|} \sum_T \frac{1}{rank_{t_p}} = \\
 &\frac{1}{5} \left( \frac{1}{6} + \frac{1}{30} \right) = \frac{1}{5} \frac{6}{30} = \frac{1}{25} = 0.04
 \end{aligned}$$

where  $t_p$  - true positives, denotes amount of relevant recommendations (match our labels),  $T$  - set of true labels and  $|T|$  - amount of true labels,  $rank_{t_p}$  - position of relevant item in recommendation results.

TABLE 4.3: Offline Evaluation of generated recommendations (English Wikipedia) on the task of predicting next 5 articles edited by user with Doc2Vec-DBOW as baseline

Model	K=50		K=100		K=200	
	Recall	MRR	Recall	MRR	Recall	MRR
Doc2Vec-DBOW	0.084	0.0162	0.1100	0.0161	0.1456	0.0162
Doc2Vec-DM	0.0702	0.0123	0.0993	0.0127	0.1350	0.0130
GraphSAGE-classification-mean	0.1021 (+21%)	0.0174	0.1408	0.0179	0.1878	0.0182
GraphSAGE-classification-meanpool	0.0885	0.0147	0.1232	0.015	0.1659	0.0151
GraphSAGE-classification-maxpool	0.0693	0.0104	0.0998	0.0105	0.1382	0.0106
GraphSAGE-link-prediction-mean	0.1236 (+47%)	0.0214	0.1702	0.0218	0.2248	0.0220
GraphSAGE-link-prediction-meanpool	0.1239 (+47%)	0.0210	0.1692	0.0214	0.2253	0.0219
GraphSAGE-link-prediction-maxpool	0.1149	0.0193	0.1594	0.0196	0.2135	0.0200
GraphSAGE-link-prediction-mean + Deep-Ranking	0.1363 (+62%)	0.0222	0.1870	<b>0.0239</b>	0.2455	<b>0.0232</b>
GraphSAGE-link-prediction-meanpool + Deep-Ranking	<b>0.1410 (+67%)</b>	<b>0.0230</b>	<b>0.1905</b>	0.0230	<b>0.2505</b>	0.0226

## 4.5 Results

We consider as our baseline Doc2Vec models, since we did not change them and trained all representations with default settings - we only selected vector size based on memory limits. As it was expected GraphSAGE models can perform much better in representation training (through adding structural knowledge into representation) than best result from Doc2Vec (Doc2Vec-DBOW): **47%** improvement in Recall, **32%** in MRR (K=50, English); **62%** in Recall and **41%** in MRR (K=50, Ukrainian). However, it is not always true for representation trained with classification task. Intuitively in this setup model preserves not enough variance and keep only minimum knowledge required for classification.

We also compared different aggregators and our results did not quite align with other researches which achieved best results with max or mean pool aggregators [Ying et al., 2018; Hamilton, Ying, and Leskovec, 2017a]. In our case *mean* and *meanpool* aggregators performed very similar with most of the cases *mean* showed better result. Whereas max-pool performed significantly worse in all experiments. Intuitively, good performance of *mean* could be explained by reach semantic properties of our input features - Doc2Vec representations, which *mean* benefits the most. In addition, models with *mean* aggregator have significantly less parameters which could lead to better generalization. This is especially likely if we recall that our model was trained in unsupervised way with respect to final task - recommendation.

Overall, our best results was achieved by combining models: GraphSAGE-link-prediction model with mean-pool aggregator for representation generation and Deep Ranking model for candidates sorting. We got **+68%** improvement in Recall and **+41%** in MRR on English edition (K=50) in comparison to Doc2Vec-DBOW baseline.

We tested our architecture against two very different editions: English and Ukrainian. We received very similar results, which indicates that our approach is stable and reliable. However, we were not able to receive much improvement with Deep Ranking trained on Ukrainian contributors due to the lack of the data - there are simply not

TABLE 4.4: Offline Evaluation of generated recommendations  
(Ukrainian Wikipedia) on the task of predicting next 5 articles edited  
by user with baseline Doc2Vec-DBOW

Model	K=50		K=100		K=200	
	Recall	MRR	Recall	MRR	Recall	MRR
Doc2Vec-DBOW	0.0832	0.0204	0.1140	0.0208	0.1390	0.0208
Doc2Vec-DM	0.0563	0.0131	0.0563	0.0125	0.0870	0.0124
GraphSAGE-classification-mean	0.0722	0.0128	0.1005	0.0128	0.1375	0.0129
GraphSAGE-classification-meanpool	0.0716	0.0121	0.0948	0.0121	0.1235	0.0123
GraphSAGE-classification-maxpool	0.0684	0.0108	0.0888	0.0110	0.1210	0.0110
GraphSAGE-link-prediction-mean	0.1352 (+62%)	<b>0.0288</b>	0.1688	<b>0.0295</b>	0.2087	<b>0.0291</b>
GraphSAGE-link-prediction-meanpool	0.1230 (+48%)	0.0280	0.1585	0.0284	0.1971	0.0287
GraphSAGE-link-prediction-maxpool	0.1162 (+40%)	0.0246	0.1458	0.0246	0.1811	0.0248
GraphSAGE-link-prediction-mean + Deep-Ranking	<b>0.1435 (+72%)</b>	0.0269	<b>0.1847</b>	0.0261	<b>0.2254</b>	0.0252
GraphSAGE-link-prediction-meanpool + Deep-Ranking	0.1388 (+67%)	0.0267	0.1763	0.0270	0.2245	0.0270

enough users to get good generalization, so by tuning the model we were able to improve either MRR or recall, not both. In contrary, from English contributors' history we were able to train the model that improve recall and MRR on **10%** in comparison with *cosine-similarity* ranking.

## Chapter 5

# Conclusion

In this thesis, we developed a technical solution for Recommendation System to recommend articles to Wikipedia editors with addressing all challenges caused by production-scale datasets (so-called Big-Data). We applied the latest research achievements in the field of Graph Representation Learning and have taken into account experience from building similar products like Recommendations on YouTube, Pinterest, eBay. Our main difference with those systems is that we cannot train our models with the supervision of previous recommendations like it was done in all of those RS due to lack of interaction history with user - 94% of Wikipedia contributors edited less than 10 articles.

We started with an overview of recommender systems and one of the most crucial parts of generating recommendation - the problem of finding good representations for recommendation items. We assumed that Graph Representation Learning that preserves both text (from articles) and structural (from links between articles) features would guarantee better performance than just text representations (word2vec, doc2vec). We introduced GraphSAGE model that is the latest state-of-the-art in Graph Convolutions for Representation Learning. We explained why this model is might be the best option for our task, especially considering that most of the other researches (considered by us) have been tested only on small lab datasets in far-from-production conditions whereas GraphSAGE was designed with production scale in mind and allowed inductive training (Chapter 2).

In our proposed solution for task of Recommending Articles to Wikipedia Editors we combined both candidate generation based on representations learned with GraphSAGE and Deep Ranking Model which aims to select only the most relevant items from generated pool of candidates and return Top-K best recommendations. The latest is the only model that was trained with supervision of previous user editions. In general, article representation were received in fully unsupervised (with respect to user interactions) fashion (Chapter 3).

On the evaluation of our system we encountered with many issues caused by big amount of data we needed to pre-process and mine in order to build our models. We developed our own toolkit to convert Wikipedia Database Dump into ready-to-training datasets. We also had to significantly rewrite the code of GraphSAGE training to enable faster data load and, hence, experimentation.

We received 47% improvement in Recall and 32% in MRR (English Wikipedia) and 62% improvement in recall and 41% in MRR (Ukrainian Wikipedia) by replacing

representations from our baseline (Doc2Vec-DBOW) with our best model (GraphSAGE-link-prediction-mean). Overall by applying our full system (GraphSAGE + Ranking) we were able to receive 68% improvement in recall and 41% in MRR (English). These results confirm our assumption about importance of structural features in Representation Learning. We were able to achieve significant improvement over just-text features even with unsupervised learning (Chapter 4).

Further research could be focused on improving current results by replacing learning task for representations. YouTube and Pinterest experience showed that the best representations for recommendations could be received when model trained with supervision from previous recommendation experience. However, it implies that there already exists some Recommender System which produces reach history of interactions - user gives feedback, by following recommended items or ignoring them. We would be able then to train on this history. Representation Learning with Graph Convolutions could show even better results in this scenario.

# Bibliography

- Babenko, A. and V. Lempitsky (2012). "The inverted multi-index". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3069–3076. DOI: [10.1109/CVPR.2012.6248038](https://doi.org/10.1109/CVPR.2012.6248038).
- Bachrach, Yoram et al. (2014). "Speeding Up the Xbox Recommender System Using a Euclidean Transformation for Inner-Product Spaces". In:
- Bahmani, Bahman, Abdur Chowdhury, and Ashish Goel (2010). "Fast Incremental and Personalized PageRank over Distributed Main Memory Databases". In: CoRR abs/1006.2880. arXiv: [1006.2880](https://arxiv.org/abs/1006.2880). URL: <http://arxiv.org/abs/1006.2880>.
- Bobadilla, J, Francisco Serradilla, and J Bernal (2010). "A new collaborative filtering metric that improves the behavior of recommender systems". In: *Knowledge-Based Systems* 23, pp. 520–528. DOI: [10.1016/j.knosys.2010.03.009](https://doi.org/10.1016/j.knosys.2010.03.009).
- Bobadilla, J. et al. (2013). "Recommender Systems Survey". In: *Know.-Based Syst.* 46, pp. 109–132. ISSN: 0950-7051. DOI: [10.1016/j.knosys.2013.03.012](https://doi.org/10.1016/j.knosys.2013.03.012). URL: <http://dx.doi.org/10.1016/j.knosys.2013.03.012>.
- Brovman, Yuri M. et al. (2016). "Optimizing Similar Item Recommendations in a Semi-structured Marketplace to Maximize Conversion". In: *Proceedings of the 10th ACM Conference on Recommender Systems*. RecSys '16. Boston, Massachusetts, USA: ACM, pp. 199–202. ISBN: 978-1-4503-4035-9. DOI: [10.1145/2959100.2959166](https://doi.org/10.1145/2959100.2959166). URL: <http://doi.acm.org/10.1145/2959100.2959166>.
- Chen, Jianfei and Jun Zhu (2018). "Stochastic Training of Graph Convolutional Networks". In: URL: <https://openreview.net/forum?id=rylejExC->.
- Covington, Paul, Jay Adams, and Emre Sargin (2016). "Deep Neural Networks for YouTube Recommendations". In: *Proceedings of the 10th ACM Conference on Recommender Systems*. RecSys '16. Boston, Massachusetts, USA: ACM, pp. 191–198. ISBN: 978-1-4503-4035-9. DOI: [10.1145/2959100.2959190](https://doi.org/10.1145/2959100.2959190). URL: <http://doi.acm.org/10.1145/2959100.2959190>.
- Dinu, Georgiana and Marco Baroni (2014). "Improving zero-shot learning by mitigating the hubness problem". In: CoRR abs/1412.6568.
- Duvenaud, David K et al. (2015). "Convolutional Networks on Graphs for Learning Molecular Fingerprints". In: *Advances in Neural Information Processing Systems* 28. Ed. by C. Cortes et al. Curran Associates, Inc., pp. 2224–2232. URL: <http://papers.nips.cc/paper/5954-convolutional-networks-on-graphs-for-learning-molecular-fingerprints.pdf>.
- Gey, Fredric C. (1994). "Inferring Probability of Relevance Using the Method of Logistic Regression". In: *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '94. Dublin, Ireland: Springer-Verlag New York, Inc., pp. 222–231. ISBN: 0-387-19889-X. URL: <http://dl.acm.org/citation.cfm?id=188490.188560>.
- Gionis, Aristides, Piotr Indyk, and Rajeev Motwani (1999). "Similarity Search in High Dimensions via Hashing". In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann



- Publishers Inc., pp. 518–529. ISBN: 1-55860-615-7. URL: <http://dl.acm.org/citation.cfm?id=645925.671516>.
- Goyal, Palash and Emilio Ferrara (2018). “Graph Embedding Techniques, Applications, and Performance: A Survey”. In: *Knowl.-Based Syst.* 151, pp. 78–94.
- Grover, Aditya and Jure Leskovec (2016). “node2vec: Scalable Feature Learning for Networks.” In: *CoRR* abs/1607.00653. URL: <http://dblp.uni-trier.de/db/journals/corr/corr1607.html#GroverL16>.
- Hamilton, William L., Rex Ying, and Jure Leskovec (2017a). “Inductive Representation Learning on Large Graphs”. In: *NIPS*.
- (2017b). “Representation Learning on Graphs: Methods and Applications”. In: *CoRR* abs/1709.05584. arXiv: 1709.05584. URL: <http://arxiv.org/abs/1709.05584>.
- Ioffe, Sergey and Christian Szegedy (2015). “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *ICML*.
- Kipf, Thomas N. and Max Welling (2016). “Semi-Supervised Classification with Graph Convolutional Networks”. In: *CoRR* abs/1609.02907.
- Le, Quoc V. and Tomas Mikolov (2014). “Distributed Representations of Sentences and Documents”. In: *CoRR* abs/1405.4053. arXiv: 1405.4053. URL: <http://arxiv.org/abs/1405.4053>.
- Liu, Tie-Yan (2009). “Learning to Rank for Information Retrieval”. In: *Found. Trends Inf. Retr.* 3.3, pp. 225–331. ISSN: 1554-0669. DOI: 10.1561/15000000016. URL: <http://dx.doi.org/10.1561/15000000016>.
- Liu, Ting et al. (2004). “An Investigation of Practical Approximate Nearest Neighbor Algorithms”. In: *Proceedings of the 17th International Conference on Neural Information Processing Systems*. NIPS’04. Vancouver, British Columbia, Canada: MIT Press, pp. 825–832. URL: <http://dl.acm.org/citation.cfm?id=2976040.2976144>.
- Malkov, Yury A. and D. A. Yashunin (2016). “Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs”. In: *CoRR* abs/1603.09320. arXiv: 1603.09320. URL: <http://arxiv.org/abs/1603.09320>.
- McNee, Sean M., John Riedl, and Joseph A. Konstan (2006). “Being Accurate is Not Enough: How Accuracy Metrics Have Hurt Recommender Systems”. In: *CHI ’06 Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’06. Montrécal, Québec, Canada: ACM, pp. 1097–1101. ISBN: 1-59593-298-4. DOI: 10.1145/1125451.1125659. URL: <http://doi.acm.org/10.1145/1125451.1125659>.
- Mikolov, Tomas et al. (2013). “Efficient Estimation of Word Representations in Vector Space”. In: *CoRR* abs/1301.3781. arXiv: 1301.3781. URL: <http://arxiv.org/abs/1301.3781>.
- Prat-Pérez, Arnau, David Dominguez-Sal, and Josep-Lluís Larriba-Pey (2014). “High Quality, Scalable and Parallel Community Detection for Large Real Graphs”. In: *Proceedings of the 23rd International Conference on World Wide Web*. WWW ’14. Seoul, Korea: ACM, pp. 225–236. ISBN: 978-1-4503-2744-2. DOI: 10.1145/2566486.2568010. URL: <http://doi.acm.org/10.1145/2566486.2568010>.
- Ricci, Francesco, Lior Rokach, and Bracha Shapira (2015). “Recommender systems: introduction and challenges”. In: *Recommender systems handbook*. Springer, pp. 1–34.
- Srivastava, Nitish et al. (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15, pp. 1929–1958. URL: <http://jmlr.org/papers/v15/srivastava14a.html>.
- Wang, Daixin, Peng Cui, and Wenwu Zhu (2016). “Structural Deep Network Embedding”. In: pp. 1225–1234. DOI: 10.1145/2939672.2939753.

- Yang, Jaewon and Jure Leskovec (2013). "Overlapping Community Detection at Scale: A Nonnegative Matrix Factorization Approach". In: *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*. WSDM '13. Rome, Italy: ACM, pp. 587–596. ISBN: 978-1-4503-1869-3. DOI: [10 . 1145 / 2433396 . 2433471](https://doi.org/10.1145/2433396.2433471). URL: <http://doi.acm.org/10.1145/2433396.2433471>.
- Ying, Rex et al. (2018). "Graph Convolutional Neural Networks for Web-Scale Recommender Systems". In: *KDD*.
- Zhang, Shuai, Lina Yao, and Aixin Sun (2017). "Deep learning based recommender system: A survey and new perspectives". In: *arXiv preprint arXiv:1707.07435*.
- Zhou, Dongyan, Songjie Niu, and Shimin Chen (2018). "Efficient Graph Computation for Node2Vec." In: *CoRR* abs/1805.00280. URL: <http://dblp.uni-trier.de/db/journals/corr/corr1805.html#abs-1805-00280>.
- Zhou, Jie et al. (2018). "Graph Neural Networks: A Review of Methods and Applications". In: 1812.08434. URL: <http://arxiv.org/1812.08434>.