INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

Improving an annotation-based artwork search engine through word embeddings

Alexander Maslew

Bachelorarbeit

Aufgabensteller	Prof. Dr. François Bry
Betreuer	Prof. Dr. François Bry, Martin Bogner
Abgabe am	2020-10-16

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 2020-10-16 Alexander Maslew

Abstract

The Artigo Project collects annotations for artworks and has a search function that can currently search for artworks using these collected annotations. The current search engine works on an exact-match basis, which means that only artworks annotated with the same annotations as searched are returned in the search result. The goal of this thesis is to improve the search results by also showing semantically relevant artworks, which do not match the exact search keyword. To achieve this word embeddings are used.

Word embedding is a very powerful mechanism that can be used for mostly all word comparison problems. It is an innovative way to enhance search results and has the potential to be improved further in the future. Its technique is to map words to word vectors which represent the meaning of the words. These vectors form a vector space, which can be used to define the similarity of the artworks. With this information, the search results can be sorted correspondingly. First, a big vector space has to be produced, which includes embeddings of words that a user will possibly type into the search field. Three existing methods for creating word embeddings are Glove, Word2Vec, and fastText. In the next step with each of these methods, for all artworks, a corresponding vector has to be calculated and put into the existing vector space. For this calculation, a custom algorithm is created that uses currently existing artwork annotations and a normalization method. Finally, the vector space needs to be searched for results. For searching, there are existing solutions available, that mostly work on the Approximate Nearest Neighbors principle. Annoy from Spotify, NGT from Yahoo and NMSLIB are examples of such solutions. These libraries are optimized for speed and search for points in word embeddings that are close to a given query point. After a comparison between the existing methods and algorithms, for the implementation, a custom Python search server is implemented which uses the vector space based search engine solution named Annoy and a corpus that has been trained with Word2Vec. Further modifications to the existing frontend and backend required to make it work are described. The result is a search engine in which users profit from improved search results and a greater chance to find what they are looking for.

Zusammenfassung

Das Artigo-Projekt sammelt Annotationen zu Kunstwerken und verfügt über eine Suchfunktion, mit der derzeit anhand dieser gesammelten Annotationen nach Kunstwerken gesucht werden kann. Die aktuelle Suchmaschine arbeitet auf einer Exakt-Match-Basis, was bedeutet, dass nur Kunstwerke, die mit exakt den gleichen Annotationen wie die gesuchten Kunstwerke annotiert sind, im Suchergebnis zurückgegeben werden. Das Ziel dieser Arbeit ist, die Suchergebnisse zu verbessern, indem auch semantisch relevante Kunstwerke angezeigt werden, die nicht mit dem exakten Suchbegriff annotiert sind. Um dies zu erreichen, werden Worteinbettungen (Word embeddings) verwendet.

Worteinbettungen sind ein sehr mächtiger Mechanismus, der für die meisten Wortvergleichsprobleme verwendet wird. Es ist ein innovativer Weg zur Verbesserung der Suchergebnisse und hat das Potenzial, in Zukunft noch weiter verbessert zu werden. Seine Technik besteht darin, Wörter auf Wortvektoren abzubilden, die die Bedeutung der Wörter repräsentieren. Diese Vektoren bilden einen Vektorraum, der dazu verwendet werden kann, die Relevanz eines Kunstwerks für ein anderes Kunstwerk zu definieren. Damit können dann die Suchergebnisse entsprechend sortiert werden. Dazu muss zuerst ein großer Vektorraum erzeugt werden, der Einbettungen von einem großen Teil der Wörter enthält, die ein Benutzer möglicherweise in das Suchfeld eingibt. Drei existierende Methoden zur Erzeugung von Vektorräumen sind Glove, Word2Vec und fastText. Im nächsten Schritt wird für jedes Kunstwerk ein entsprechender Vektor berechnet, der die gleiche Form wie die Word embedding Vektoren hat damit man später beide vergleichen kann. Für diese Berechnung wird ein neuer Algorithmus erstellt, der aktuell existierende Kunstwerk-Annotationen und eine Methode zur Normalisierung verwendet. Abschließend muss der Vektorraum nach Ergebnissen durchsucht werden. Für die Suche gibt es bereits existierende Lösungen, die eine gute Performanz erzielen und meist nach dem Prinzip des Approximate Nearest Neighbors arbeiten. Annoy von Spotify, NGT von Yahoo und NMSLIB sind Beispiele für solche Lösungen. Diese Bibliotheken suchen nach Punkten in Vektorräumen, die nahe an einem bestimmten Abfragepunkt liegen. Für die Implementierung wird ein Python-Suchserver implementiert, der Annoy und einen mit Word2Vec trainierten Korpus verwendet. Weitere Modifikationen am bestehenden Frontend und Backend, die erforderlich sind, damit alles funktioniert, sind beschrieben. Das Ergebnis ist eine Suchmaschine, bei der die Benutzer von verbesserten Ergebnissen und einer größeren Chance, das Gesuchte zu finden, profitieren.

Acknowledgments

First of all, I want to thank Prof. François Bry for his supervision and for the opportunity to work on this topic at the chair for Programming and Modeling Languages. I also want to thank my mentor M.Sc Martin Bogner, who provided me with new ideas and guided me throughout this thesis. Meetings with him every week helped me point this bachelor thesis in the right direction. Finally, I want to thank my friends and family who supported me and participated in the evaluation part of this work.

Contents

CHAPTER 1

Introduction

The ARTigo Project collects annotations for digitally saved artwork reproductions using several games, in which players describe artworks with annotations (also known as tags or keywords) to them. Such games are often referred to as Games with a Purpose (GWAP). Their goal is to get artwork tags and they achieve this by motivating players to play the games. ARTigo's collection mostly contains artworks made by European artists in the time period starting at the end of the 18th century and ending at the beginning of the 20th century.

ARTigo's goal is to have a semantic artwork search engine, which can find artwork-images using annotations, collected with GWAP. Currently, ARTigo has a search engine that works on an exact-match basis, which means that only artworks can be found, which really are annotated with the searched annotation. It is based on the search engine platform Apache Lucene/Solr, which also powers the search and navigation features of many of the world's largest internet sites [27].

Search engines are a specialized type of software for retrieving information stored on a computer system or network [39]. Generally the user (searcher) makes a query for content that meets a certain criterion (usually one that contains certain words and phrases), which results is a list of points that meet, in whole or in part, the requested criterion. Search engines are the most visible Information Retrieval (IR) systems.

However, ARTigo's current search engine has several limitations, that lead to limited results. Currently, similar images with related annotations, which are most likely relevant can not be recognized as such. Additionally, when the user searches for words, that have never been used as annotations on artworks, no results can be found.

For example, if we have two images, the first one annotated once with "tree" and the second also tagged once, but with "forest". When the user now searches for "tree", only the first image will be found, because a connection between "tree" and "forest" is not known to the search engine.

This bachelor thesis reports on adding such connections between all ARTigo artworks using a deep learning technique named word embeddings and existing artwork information like annotations, artist name, etc. This improves the current search engine by adding the ability to also find semantically similar artworks.

This thesis starts with a representation of the related work in the field of word embeddings and their search engines. Then a concept of the vector space based search engine is elaborated and its implementation is described. Finally, suggestions for further research

and additional technical details required for the implementation and maintenance are provided.

CHAPTER 2

Related Work

2.1 Word embeddings

The technique of mapping a word to a vector that represents its semantic meaning is called word embedding. In Natural Language Processing (NLP), an embedding refers to a vector that represents the meaning of a word. The embeddings usually have high dimensionality, meaning each embedding is made out of a vector that contains hundreds or in some cases even thousands of scalars. A vector is a one-dimensional sequence of numbers (often called scalars) that describes how much of something there is. Inside the vector space, semantically related words are represented by similar embeddings and vice-versa. Also, depending on the properties of the underlying algorithm used to create the word embeddings, mathematical operations like addition and subtraction can be applied and this ability can be used in finding relationships between vectors.

2.2 Word embedding Generators

Word embeddings can be generated using existing methods. Three well-established algorithms used for generating word embeddings are Word2Vec, GloVe, and FastText, which will be discussed in the following:

2.2.1 Word2Vec

Word2Vec was created and published by a team of researchers at Google in 2013 and has since become a popular method to generate word embeddings. Word2Vec translates words into vectors and its model is predictive. It uses neuronal networks to train the vectors and for assigning words to vectors Word2Vec uses the word context. With this approach words that occur in similar contexts will have similar embeddings.

For example, if we have the following two sentences:
The **kid** said he would grow up to be superman.
The **child** said he would grow up to be superman.

The words **kid** and **child** will have similar word vectors due to a similar context. When iterating through a large enough corpus a lot of sentences will have the words kid replaceable with child. It is also possible to capture different degrees of word similarity. Calculations like ”Brother” + ”Woman” - ”Man” can produce a result vector that is located closest to the vector of ”Sister” [19].

Word2Vec offers two important model architectures. The first one is Continuous bag of words (CBOW) and the second is Skip-Gram.

In CBOW the output is the target word (jumps) and is predicted from the context:

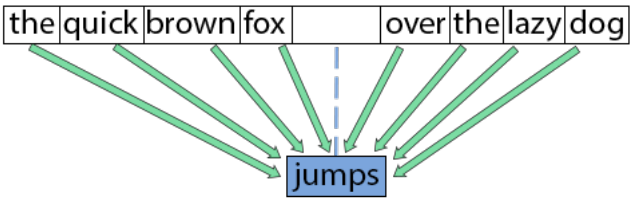

Figure 2.1: CBOW method.

In Skip-Gram the outputs are the context words and they are predicted from the target word (jumps)

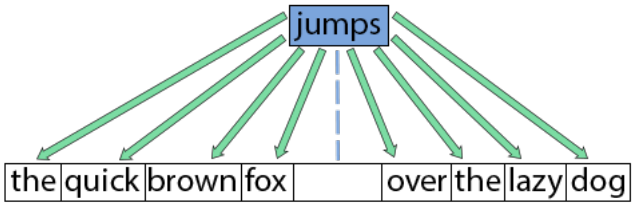

Figure 2.2: Skip-Gram method.

CBOW is superior to Skip-Gram in evaluating frequent words, but Skip-Gram evaluates rare words more precisely than CBOW [42]. A good evaluation of rare words is crucial for many use cases and for this reason the Skip-Gram variant of Word2Vec is more commonly used. However, Skip-Gram is only favorable when working with a large Corpus and is slower compared to CBOW.

Important parameters for the training are content-window size and number of vector dimensions. The content-window determines how many words before and after the target word will be included in the target word context. Every additional vector dimension allows more space to save word details.

The accuracy of Word2Vec can be improved by:

- Choice of model architecture

  For large corpus and higher dimensions - slower Skip-Gram

  For a small corpus - faster CBOW

- Increasing the training dataset size
- Increasing the number of vector dimensions. Typically set between 100 and 1,000.
- Increasing the windows size. The recommended value is 10 for Skip-Gram and 5 for CBOW. [42]

2.2.2 GloVe

GloVe stands for Global Vectors for Word Representation and was created at the University of Stanford. GloVe's approach for creating word vectors is different from Word2Vec. Instead of capturing co-occurrence one window at a time (like Word2Vec), GloVe tries to capture overall statistical counts of how often co-occurrence appears for the whole corpus. To achieve this GloVe's model is count-based and instead of neural networks, it uses a co-occurrence Matrix [31].

GloVe is easier to parallelize than Word2Vec, because of it's count-based approach and this improves the ability to train over big data. But at the same time GloVe consumes a considerably higher amount of system memory while training, due to its co-occurrence matrix.

The GloVe co-occurrence matrix is as long and as wide as the vocabulary size. It contains only numeric values equal to or greater than zero and is used to calculate vectors. Each time words occur together within a specified window-size in the corpus they count as related and their corresponding entry in the co-occurrence matrix is incremented. The window-size works just like in Word2Vec. It specifies how many words before and after a target word count as related.

If for example, we have a window size of 3, the corpus "this is a GloVe matrix test" and one iteration over the corpus, a co-occurrence matrix could look similar to this:

	this	is	a	GloVe	matrix	test
this	0	1	0	0	0	0
is	1	0	1	0	0	0
a	0	1	0	1	0	0
GloVe	0	0	1	0	1	0
matrix	0	0	0	1	0	1
test	0	0	0	0	1	0

Table 2.1: GloVe co-occurrence matrix

With every iteration through the corpus, modifications to the co-occurrence matrix and the accuracy are done. After the last iteration finished calculating, the co-occurrence matrix is used to calculate final result vectors which are then saved to a file before the GloVe algorithm is halted.

According to test results from the creators of GloVe, the recommended amount of iterations depends on the corpus size and is usually between 15 to 25. The recommended content-window size is 10. Important parameters for achieving better accuracy are the number of dimensions, iterations over the corpus, and content-window size [31].

Also, according to the authors of GloVe, in a comparison between Word2Vec and GloVe using the same corpus, vocabulary, window size, and training time, GloVe consistently outperformed Word2Vec. It achieved better results faster and obtained the best results irrespective of speed [31].

2.2.3 fastText

FastText was created by Facebook's AI Research lab in 2015 and it is maintained by them since. It is based on Word2Vec but has several improvements not present in Word2Vec and GloVe. The main improvement of fastText is that it improves word embeddings by enriching the word vectors with subword information. Instead of learning vectors for words directly, fastText represents every word as an n-gram of characters. For example, the word "artificial" with n=3, would result in (ar, art, rti, ifi, fic, ici, ial, al). This should help with the representation of words that include suffixes or prefixes, especially for large vocabularies with many rare words [3].

After the n-grams have been calculated modified algorithms are available for training of the word embeddings (which add support for word n-grams to CBOW and Skip-Gram).

2.3 Vector space search engines

After a vector space containing all artwork identifiers is successfully generated and a vector that corresponds to the user search input is fetched from existing word embeddings, the search for related artworks can finally begin.

To find artworks near to the input vector of a search query, a method for comparing existing vectors to a target vector is necessary. Methods that fulfill this goal are distance functions. Depending on the exact function used they either measure the distance or angle between two vectors to determine their similarity. The output of a distance function is usually a number between 0 and 1, where values closer to one are more similar.

Vector space search engines are using distance functions and are usually doing nearest neighbor searches. They are quite complex because they can handle large volumes of data in high-dimensional vector spaces and at the same time aim to be as efficient and consistent with the results as possible.

2.3.1 Existing solutions

For searching in vector spaces, there are many existing libraries available. The most used and well-known library is Annoy. But according to most openly available benchmarks in multiple scenarios, two other libraries named NGT and NMSLIB should be slightly faster than Annoy [21]. The best performing library currently available, according to Facebook AI Research, is Facebook's Faiss algorithm using the optional GPU-based implementation [7]. Annoy, NGT, and NMSLIB are based on Approximate Nearest Neighbors (ANN) search. Faiss can use both Fixed (which searches for all points within a given fixed distance from a specified point) and ANN search. But to keep the implementation for ARTigo simple we will focus on CPU-based approaches only.

Annoy is created and maintained by Spotify and used by them internally for music recommendations. In comparison to other vector space search engines and according to Github, Annoy currently has the largest community with over 1.000 Github Repositories that depend on it. One advantage that sets it apart from others is that it uses static files as indexes, which are loaded into memory using "mmap" and can also be shared across multiple processes, which improves scalability.

Annoy uses Tree-based Indexing, which means that the entire search space is recursively divided into hierarchical subspaces so that the search space forms a tree structure [36]. Given a search query, annoy efficiently finds the subspaces by descending from the root node to the leaf nodes in the tree structure and then obtains its search results by scanning only neighbors belonging to the subspaces [5].

The following distance functions are supported: angular, euclidean, manhattan, cosine, hamming, and dot (inner) product.

NGT stands for Neighborhood Graph and Tree for Indexing High-dimensional Data. It has been created by Yahoo Japan.

NGT's indexing method is Graph-based and uses a neighborhood graph, where every node is connected to its, calculated by distance functions, nearest neighbors. Given a search query, NGT retrieves the first nearest neighbor and then uses it to recursively lookup the next nearest neighbor until the requested amount of results is reached. The exact algorithm used by NGT is the so-called ANNG (approximate k-nearest neighbors) [36].

Supported distance functions are euclidean, angular, cosine, manhattan, hamming, and jaccard.

NMSLIB stands for Non-Metric Space Library. It was created by Leonid Boytsov and Bilegsaikhan Naidan and has recently gained popularity. In particular, it has become a part of Amazon Elasticsearch Service [24]. The search algorithm used is Hierarchical Navigable Small World graphs (HNSW) [18], which according to benchmarks, is one of the leading methods for approximate neighbor searches in terms of speed. HNSW is fully graph-based and incrementally builds a multi-layer structure consisting of a hierarchical set of proximity graphs (layers), that create nested subsets of the stored elements [18].

NMSLIB also comes with a query server, which depending on the project, might be useful. Supported distance functions are euclidean and cosine. According to the official NMSLIB Github page [24], more distances should be available, however, an inspection revealed that they are currently not fully supported.

2.3.2 Distance functions

Vector space search engines use distance functions to determine the similarity between two vectors. Euclidean, angular, cosine, and manhattan distance functions are often already implemented and available to choose from. Choosing the right distance function is very important because it can drastically change the search results. Distance functions can be categorized with L^p norms, where p is a value greater than zero. L^p norms are a kind of measure of the size of a mathematical object [33].

In high-dimensional space the data becomes sparse, and traditional indexing & algorithmic techniques fail from an efficiency and/or effectiveness perspective [10]. The reason behind this is the phenomenon called the curse of dimensionality, which arises in high dimensional space and causes the ratio between the nearest and farthest points to approach 1:1. This results in a complete inability to distinct some points, which ultimately leads to poor results [2]. The curse of dimensionality phenomenon is more noticeable for distance functions with a higher p number and since word embeddings are usually stored with high dimensionality, choosing a L^p space with a low p number is crucial.

Another important factor when working with high dimension data is that the distance measurement between two points should be more accurate when the angle between the two points is used for measurement, rather than to measure the distance between the two points in a straight line [23]. For this reason, the euclidean distance (also known as L^2 norm), should perform worse when comparing two sparse vectors in high dimensional space than for example cosine distance or manhattan distance (also known as L^1 norm).

Formulas used to measure the distance between two vectors X and Y with k dimensions:

- Euclidean distance: $\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$

- Cosine distance: $1 - sim(A,B)$, where cosine similarity: $sim(A,B) = cos(\theta) = \frac{A \cdot B}{|A| \cdot |B|}$ and θ is the angle between the vectors A and B. As the cosine distance between the vectors increases, the cosine similarity, or the amount of similarity decreases, and vice versa. Thus, vectors closer to each other are more similar than vectors that are far away from each other [22].
- Manhattan distance: $\sum_{i=1}^{k} |x_i - y_i|$. The sum of (absolute) differences of the vectors.
- Angular distance: $\frac{\cos^{-1} \cdot (cosine\ similarity)}{\pi}$. A problem with cosine similarity is that when the angle between two vectors is small, the cosine function delivers results very close to 1 and some precision can be lost. Angular distance yields more different similarity values for an angle close to 1 [4].

The following graphs illustrate measuring vectors in a two-dimensional space:

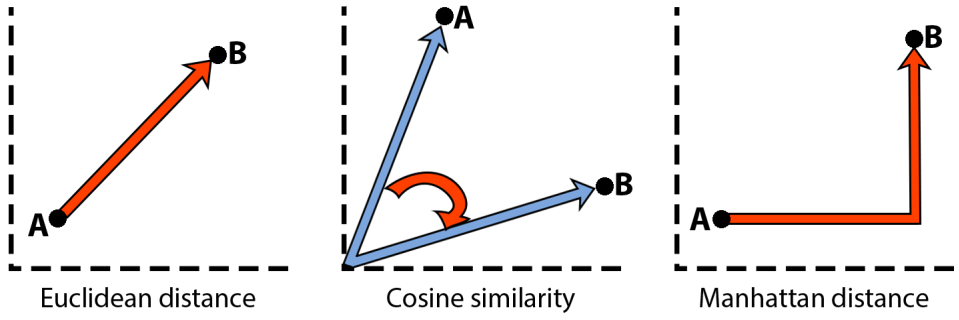

Figure 2.3: Distance functions.

2.4 React

React (or React.js) is an open-source JavaScript library for building user interfaces and was initially released by Facebook's web development team in 2013. Today React is one of the best choices for building modern web applications, because it has a slim and easy to use API, an ecosystem with many existing frameworks and extensions, and a great community. Developers using React profit from the convenient JSX syntax and the declarative style of programming, which results in less code.

JSX syntax
The JSX syntax is an extension of the JavaScript language and allows the writing of HTML code directly into JavaScript [32] without any additional methods. It converts HTML tags into react elements, which are later used for rendering [12]. The JSX syntax makes it easier and more convenient to write React applications, but it is possible to use React without it [13].

DOM rendering engine
Elements describe what we see on screen and are the smallest building blocks in React. The Document Object Model (DOM) or "real DOM" is a programming interface for HTML and

XML documents. It provides the structural representation of all nodes in an HTML document in a tree structure and a way for JavaScript to interact with every node [20]. To make changes to the user interface in a browser, developers have to edit the browser's DOM. But editing the DOM frequently can result in a significant impact on performance. For this reason, React uses a virtual DOM, which is an abstraction of the real HTML DOM. The virtual DOM is lightweight and improves the performance significantly because it ensures that only the minimum of required operations on the real DOM is made by calculating the best possible way to make these changes [16]. For the calculation first, a new virtual DOM is created and compared to the existing one. Then the best possible edit is calculated and the minimal changes are made to the real DOM.

2.5 Node.js

Node.js is a software platform built on Google Chrome's V8 JavaScript run-time and it uses an event-driven, non-blocking input-output model, which enables it to facilitate several requests at the same time. This improves scalability and makes Node.js ideal for real-time data-intensive applications. Since Node.js is a run-time environment, it can be used for both front-end and back-end development [15]. The use of JavaScript brings several benefits to the developers:

- The JavaScript language is naturally asynchronous, which means that each operation is only started after the preceding one is completed. This adds the ability to execute another process while waiting for the current one to complete.
- A huge base of web developers have already used the JavaScript language at some point on the front-end and are familiar with it.
- Developers can share code between the front-end and back-end since JavaScript is also widely used for front-end development.
- Native support for JavaScript Object Notation (JSON), which is a very popular lightweight data-interchange format. JSON is easy for humans to read and write [17].

Node Package Manager
The free Node Package Manager (npm) is the default package manager for Node.js and has recently become the center of JavaScript code sharing. It offers a lot of ready-made solutions developers can easily add to and use in Node.js. Currently, there are more than one million packages published in npm, which makes it the largest software registry in the world [25]. It is important to note that packages are usually not deleted from npm and because of that some are outdated or not maintained anymore and unusable with the current version of Node.js. A study from 2019 showed that 61% of packages did not publish a release in the last 12 months [37]. Also, the registry can contain low quality, insecure, or malicious packages [29]. But still, they are plenty of good up-to-date packages to choose from.

CHAPTER 3

Concept

3.1 Idea

The idea is to improve ARTigo's current search engine by implementing a completely new solution, based on word and artwork embeddings.

Currently, the ARTigo search engine works on an exact-match basis, which means that it can find artworks, which have been annotated with exactly the same search term. It can not find artworks with similar but not the same annotations or in other words, artworks that have not been annotated with the exact search term. This bachelor thesis addresses this limitation and presents a solution using word embeddings in which users will profit from an overall better search experience with improved results and if they are searching for something specific - a greater chance to find what they are looking for. The use of word embeddings makes sense for the ARTigo search engine because each artwork and user search input can be represented by a vector and existing approximate nearest neighbor solutions can be used to compare these vectors and find matching artworks quickly.

The first step is to generate vectors for a large collection of words (word embeddings), which will provide a wide variety of words, that a user could possibly type into the search field and a corresponding vector for every word that represents the words meaning. The process of generating word embeddings is called training and uses specialized methods that require a corpus, which is a text containing a large collection of words or sentences. There are two ways that come in mind for choosing a corpus for training purposes. The first option is to use existing artwork annotations and the second approach is to use a large collection of sentences which for example can be acquired from the Internet using a Wikipedia or Twitter dump. It is important to have a good distinction between different words so that the search engine can work properly. To achieve this the corpus size needs to be large enough because the accuracy of words depends on the corpus size. A larger collection of words / sentences typically results in higher accuracy and improved distinction of fundamentally different words. To estimate the meaning of each word with high accuracy, as many as possible distinct words in the same context are required. This means that a larger corpus with an extensive vocabulary will almost always achieve higher word accuracy. Another important factor is that the user has no limitations on what to search and therefore it is crucial to have a large enough vocabulary to increase the chance of finding a vector that corresponds to the requested search term in the word embeddings file. Also, a

large enough vocabulary is crucial for defining more semantically similar words.

The use of existing ARTigo artwork tags to create a corpus may not be a good choice, because the current amount of unique annotations is only about 317 thousand which is considered not enough in natural language processing and a lot of them are used only once, which would make it impossible to define a context to calculate their meaning. On the other hand, a Wikipedia corpus for example has a vocabulary size in the millions and since each word is almost always contained in a sentence, a good enough context is typically given. Therefore word embeddings in this bachelor thesis will be created using an external corpus.

3.1.1 Choice of existing solutions

Since the process of generating word embeddings is quite complex (see section 2.2 of this thesis), implementing a new solution from scratch is not worth it and a choice of three existing methods Word2Vec, GloVe, and fastText has been made.

Word2Vec has been chosen because it is the most popular and known method. GloVe was chosen because its model is fundamentally different than Word2Vec and according to the creators of GloVe in a comparison between Word2Vec and GloVe it should be outperforming Word2Vec. And finally, the choice for fastText is made, because it is a modified version of Word2Vec with a slightly different approach and several, according to the authors, improvements.

Word embeddings are required for the ability to map a user search input to a vector. But creating a vector from an artwork is different because it is not just a simple word. The idea is to create a custom algorithm for use with the ARTigo artworks, which utilizes a combination of existing artwork annotations, word embeddings, and normalization techniques to build artwork embeddings.

Finally, a vector that corresponds to the user search input is fetched from the word embeddings and used to find matching vectors from the artwork embeddings using the in section 2.3 of this thesis, discussed methods Annoy, NGT, and NMSLIB for searching through word embeddings. The choice for Annoy was made because just like Word2Vec it is the most well-known and popular method in its category. NGT was chosen because its indexing approach differs from Annoy. And finally, NMSLIB was chosen, because according to benchmarks it should be one of the leading methods for approximate nearest neighbor searches in terms of speed.

Since all discussed methods just like most natural language processing programs use the programming language Python for implementation a connection between the existing ARTigo Node.js backend server and Python is required. The communication can be achieved using a simple command-line interface (CLI) or a socket communication. However, the use of a CLI is not a good option, because a CLI would load a Python script once and then terminate it after a result is received. This means that for every request all the required data (word embedding indexes, artwork embedding indexes, etc) would have to be loaded into memory before result artworks can be found and returned which results in a very long response time. The use of a socket communication can fix this problem because it allows Python to load all required files into memory once and then wait for a socket request to fulfill. There are many existing socket communication libraries available and the choice of ZeroMQ has been made because it is well-known, open-source, and offers simple implementation for both Node.js and Python.

3.2 Design

The current ARTigo website was initially created back in 2012 using the Extensible Hypertext Markup Language (XHTML), which is an extended version of the widely used

Hypertext Markup Language (HTML). Back then it was a suitable solution, but today the web framework React is state of the art for building user interfaces or UI components. The new interface implemented in this bachelor thesis does not aim for great design and is mostly built for proof of concept. Therefore most HTML and CSS code responsible for the design is taken from the current ARTigo website and applied to the new React frontend implementation.

3.2.1 React Frontend Search Interface

The search bar is located at the top right corner and is available while navigating through all pages on the ARTigo website. When the user types a search term into the search bar and clicks the search button a new page is loaded. It includes a list of search results and every result is a table with three columns. On the left, it has a counter for how often an Image was annotated with the requested search term. In the middle, the corresponding image is displayed. And on the right side of the image additional information about the artwork is displayed, including the artist, title, location, and creation date.

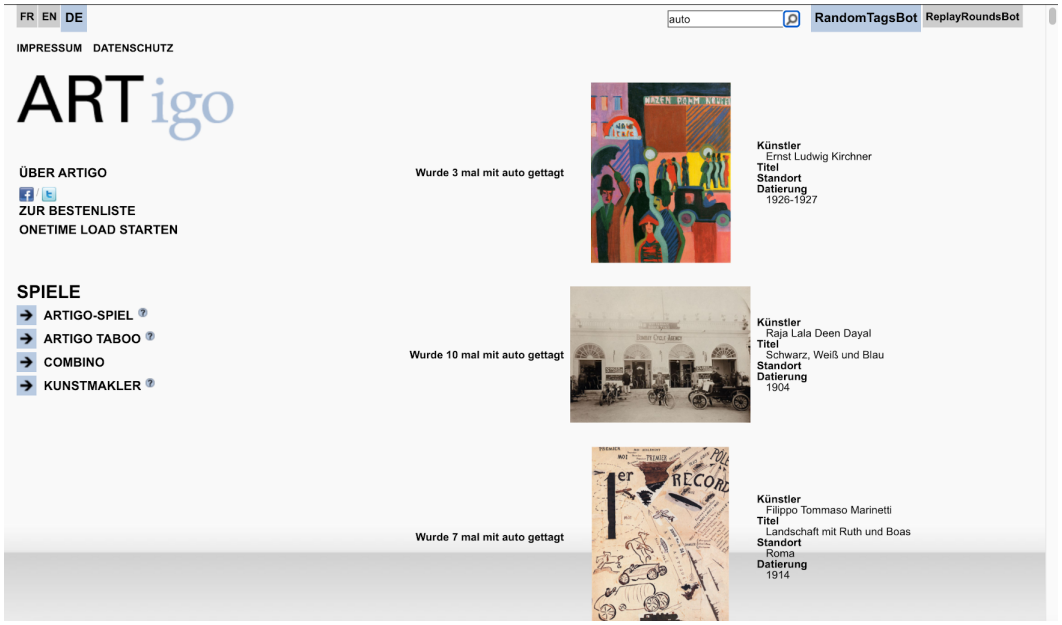

Figure 3.1: Search results page

It also includes a detail page that shows up after clicking on an Image. The detail page shows a larger image of the corresponding artwork, that fills the whole page. To the right of the image more additional information is displayed, including all annotations of the artwork, which can be used to understand why the shown artwork was selected by the search engine.

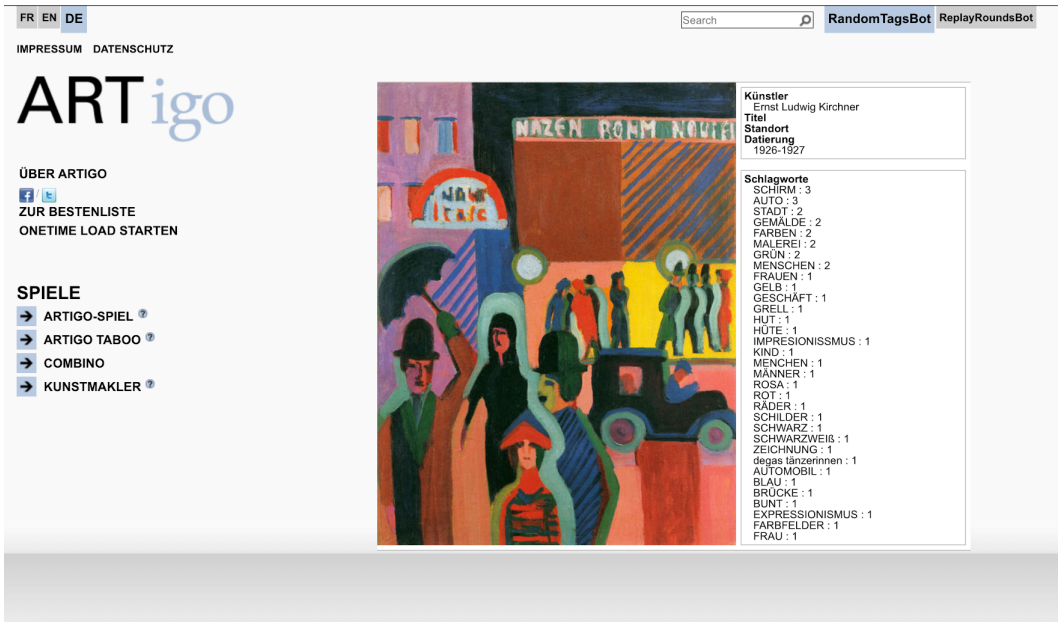

Figure 3.2: Detailed search results page

CHAPTER 4

Implementation

4.1 Front-end

The front-end is implemented using React which is a well-known open-source JavaScript library for building user interfaces made by Facebook and community [26]. React has been chosen because the combination of React and Node.js (back-end solution) is proven to be a very solid [38] and because the current development implementation of ARTigo (which is not yet available online) uses React for its front-end.

The implementation consists of the following parts:

- Search-header object with a field and button which is permanently visible on every page. This is achieved by adding the object to the always visible main page which also includes the content and other navigation panels. It looks like this:

- Search Page responsible for receiving data from the back-end and once received for the rendering of the search results.

- Detail Page for additional Information with a larger image just for one selected artwork. The required data is fetched with a SQL query.

Search results are retrieved from the back-end as a list of objects. Every object represents an artwork and contains further unique properties, which are rendered for display using two classes.

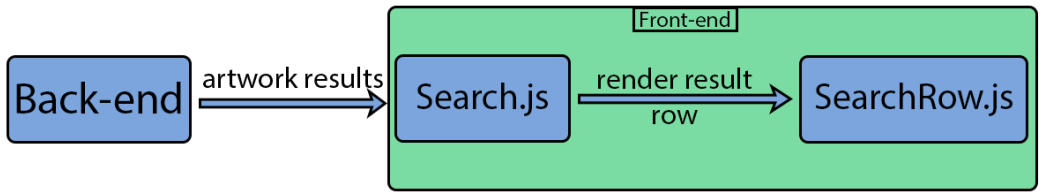

Figure 4.1: Front-end classes Search and SearchRow.

To generate HTML code from JavaScript, which a browser can understand React uses a technique called rendering [14], which was further described in section 2.4 of this bachelor thesis. The results from the back-end are rendered in the Search page (Search.js class) where rendering is completed by using the SearchRow.js helper class, which renders every search result-row individually.

4.2 Back-end

The back-end is implemented using Node.js, which is an open-source, cross-platform, JavaScript run-time environment that executes JavaScript code using Google Chrome's V8 engine outside a web browser [8]. Node.js is a very convenient solution for many web-developers because most of them are already familiar with the JavaScript programming language, which is frequently used for front-end development. Also, the currently existing ARTigo project already implements Node.js for the back-end. Node.js was further described in section 2.5 of this thesis.

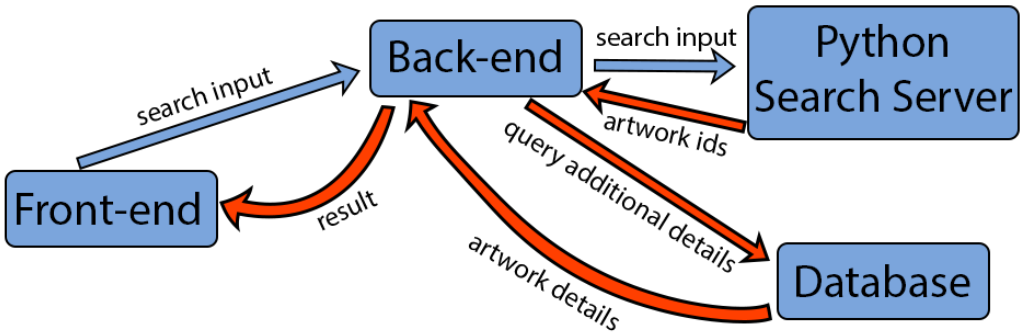

Figure 4.2: Communication between the back-end, front-end, search server and database.

The main back-end component is responsible for the communication between the front-end and the search server. For the transmission among front-end and back-end existing Node.js socket is used and for the transmission between back-end and search server, a high-performance asynchronous messaging library called Zero-MQ [28] is used.

Zero-MQ was chosen because it is industry-leading, lightweight, and available for both Node.js and Python, which means that the same library can be used for both back-end and

search-server components. When the user starts a new search:

1. Input gets forwarded to the Python search server via the back-end.
2. Artworks IDs are received at the back-end and the database is queried for additional artwork details like title, location, date, etc.
3. Final results including image URL's are transmitted to the React front-end for display.

4.3 Generation of word embeddings

To generate Word embeddings first a corpus is created and then trained using the three in chapter 2.2 discussed methods Word2Vec, GloVe, and fastText. Finally, in the next section, all resulting word embeddings are compared and the best suitable one for ARTigo is chosen.

4.3.1 Selecting a suitable corpus

A corpus can be created using different types of text data, like for example crawling random websites or downloading a large number of posts from social networking sites. It is also important to have a large enough corpus, since the accuracy of Word2Vec, GloVe and fastText increases with the corpus growth [34].

Wikipedia dumps are used frequently in Natural Language Processing because they contain a large variety of different words from easy to complex and contain very little to none grammatical or other writing errors compared to for example a Twitter corpus. Also, Wikipedia offers the ability to download huge dumps of data containing millions of articles, available for a wide variety of languages and thematically, since Wikipedia is an encyclopedia, it spans over a wide arrange of topics, just like ARTigo artworks do. For these reasons, a Wikipedia corpus should be very suitable for an artwork-based search engine.

4.3.2 Preparing corpus

The downloaded Wikipedia dump can not be used directly to train a vector-space, because it is stored as a compressed XML-file and contains information like identifiers, page numbers, and other Wikipedia-related Information that is embedded in the file with XML-Syntax. After extracting the downloaded dump, it needs to be prepared by cleaning and then pre-processing it. This means removing all Wikipedia markups, remove punctuation, dates, numbers, etc. and finally writing one sentence per line to a single text file. Additionally, since ARTigo annotations are all stored uppercased, to improve compatibility all words need to be uppercased too. This will for example remove duplicate words like "car" and "Car", that have no use in ARTigo, by replacing them both with "CAR".

To prepare the Wikipedia dump existing methods are available. One option is to use different tools for cleaning and pre-processing. For example wiki-extractor for cleaning and a library like Microsoft's BlingFire tokenizer for pre-processing. But a more revised option would be to use just one tool, that is specifically designed to create a corpus from the Wikipedia dump. A preferred choice for creating a Wikipedia based corpus is the topic modeling library named Gensim because it includes a class made just for this task and is proven to work very reliably by many users.

4.3.3 Training corpus

As explained in the related work section of this thesis, methods for generating word embeddings are quite complex. A well-known existing solution, that is proven to work reliably,

would be more viable, than creating a new one from scratch. Detailed instructions on how to train word embeddings can be found in the appendix section of this thesis.

4.4 Comparison between Word2Vec, GloVe and fastText

All performance measurements have been taken on a Google cloud instance with 8x AMD EPYC 7B12 CPUs, 64 Gigabytes of system memory, and SSD-based storage.

4.4.1 Resources and time needed to train corpus

The Wikipedia corpus in the German language with a vocabulary size of 2.15 million (unique words) generated with Gensim resulted in a file size of 7.28 GB. Training word embeddings on that corpus for 300 dimensions using recommended hyper-parameters for all methods (apart from the additional test of 15 epochs for Word2Vec & fastText) required the following:

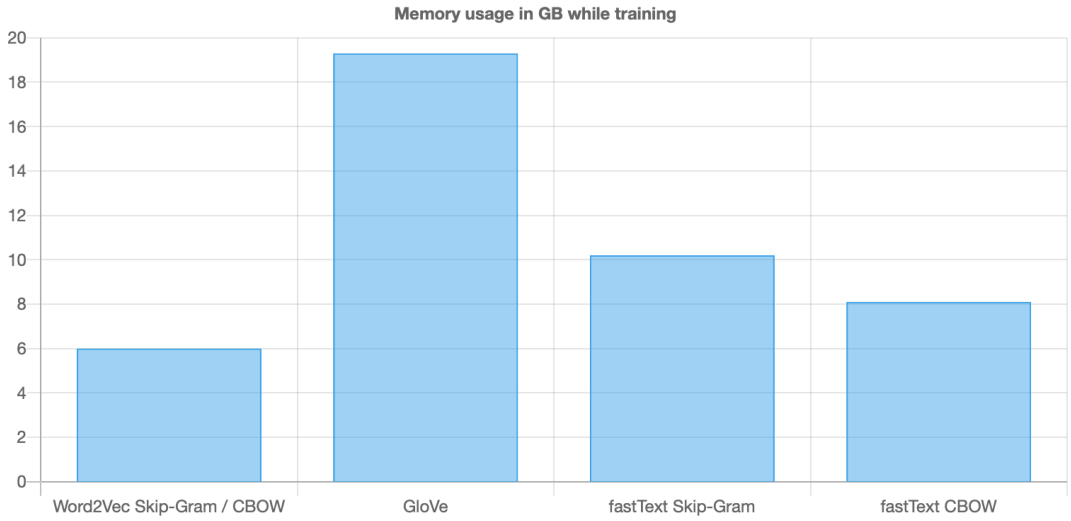

Figure 4.3: Memory usage while training with Word2Vec, GloVe, and fastText.

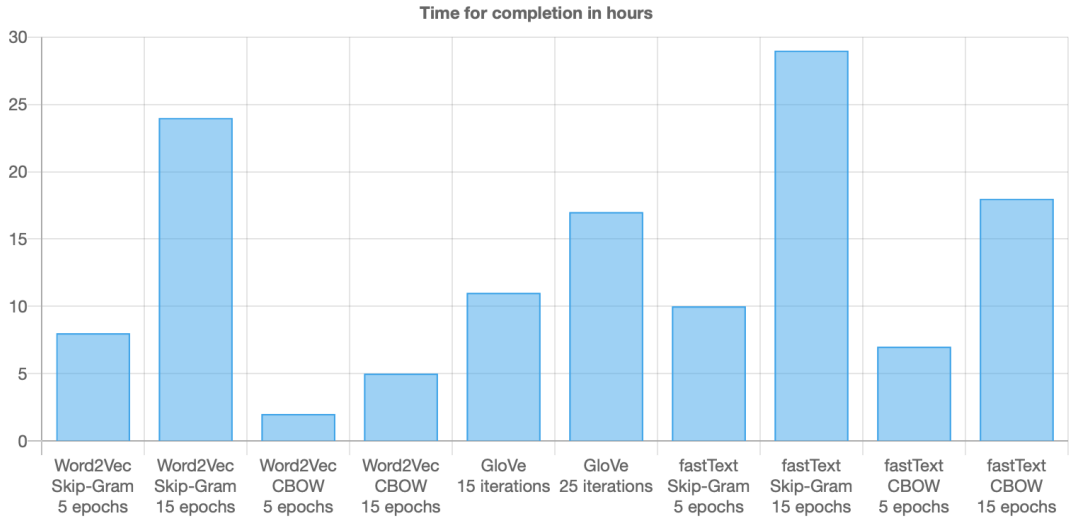

Figure 4.4: Required time for training with Word2Vec, GloVe, and fastText.

The result word embeddings file is around 6.1 GB in size and contains the same amount of 2.15 million words (also known as vocabulary size) for every method tested. Another important note is the high memory usage of GloVe (see Figure 4.3), which would require additional swap space, if trained on a machine with not enough memory. Swap usage would result in a significant increase in training time, due to additional I/O operations.

4.4.2 Result evaluation

To evaluate the generated word embeddings, features are visualized using Principal Component Analysis (PCA), which is a method that reduces the number of dimensions in a high dimensional vector space but keeps main features (principal components). PCA is probably the most popular multivariate statistical technique and it is used by almost all scientific disciplines [1]. A method like PCA is used because it is not possible for humans to visualize 300 dimensions, that were created while training the corpus. With PCA, the dimensions were reduced to just two and then plotted using a python library called “matploblib”. The following visualizations were created by using word embeddings, that have been trained using a German Wikipedia corpus. Recommended values for Word2Vec (Skip-Gram with 5 iterations), fastText (Skip-Gram with 5 iterations), and GloVe (25 iterations)are used. On the left side are visualizations for currencies, in the middle for capitals, and on the right for genders.

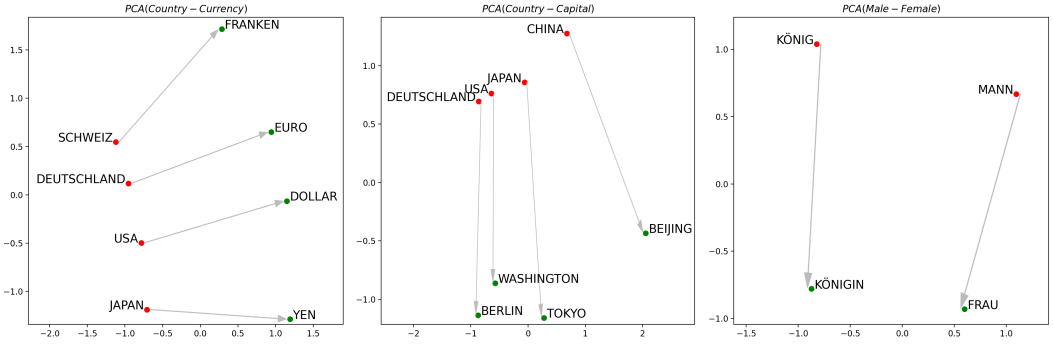

Figure 4.5: PCA visualizations for Word2Vec.

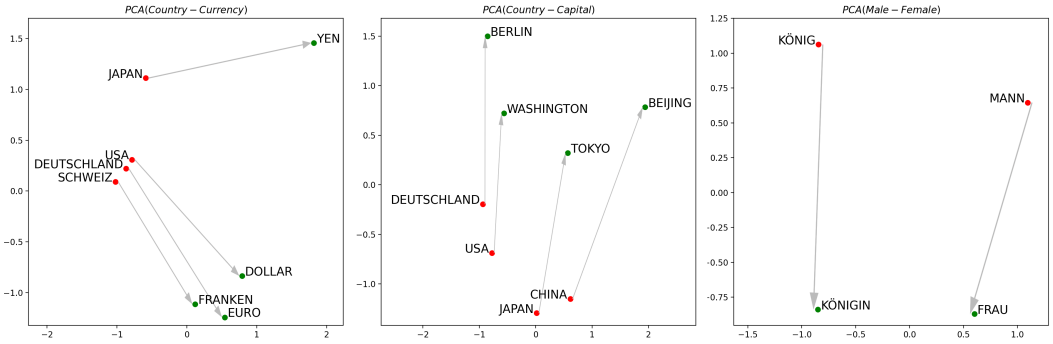

Figure 4.6: PCA visualizations for fastText.

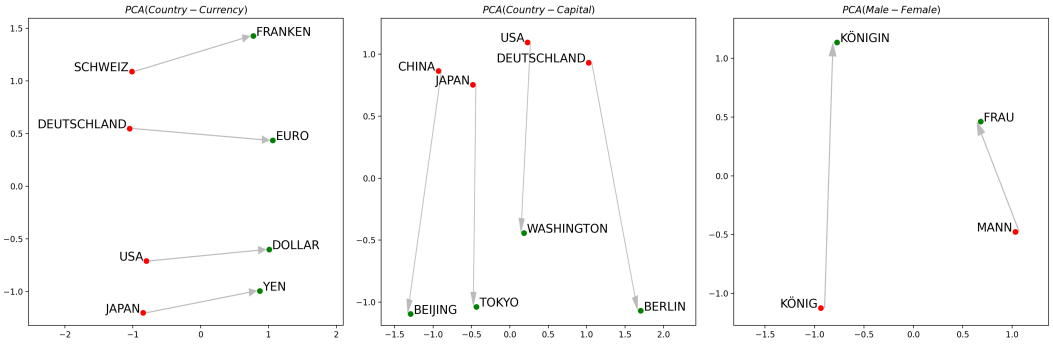

Figure 4.7: PCA visualizations for GloVe.

All three methods clearly portray a connection, but it is difficult to decide which one is better by just looking at these visualizations.

The next approach is to search for neighbors of common words and compare them. This can also be achieved with PCA, but libraries based on approximate nearest neighbors will not provide a good visialization, because they are only approximate and the different

distance functions would have to be evaluated as-well. This means that every request can yield slightly different results, which is not optimal when comparing the algorithms with PCA. For this reason a fixed approach is needed and since good performance is not necessary for visualization, the Gensim model can be used which includes a function for fixed neighbors search that corresponds to the word-analogy and distance scripts in the original Word2Vec implementation [9]. The following graphs visualize the nearest neighbors of Word2Vec, fastText and GloVe using PCA for the three German words “Auto”, “Computer” and “Wetter”.

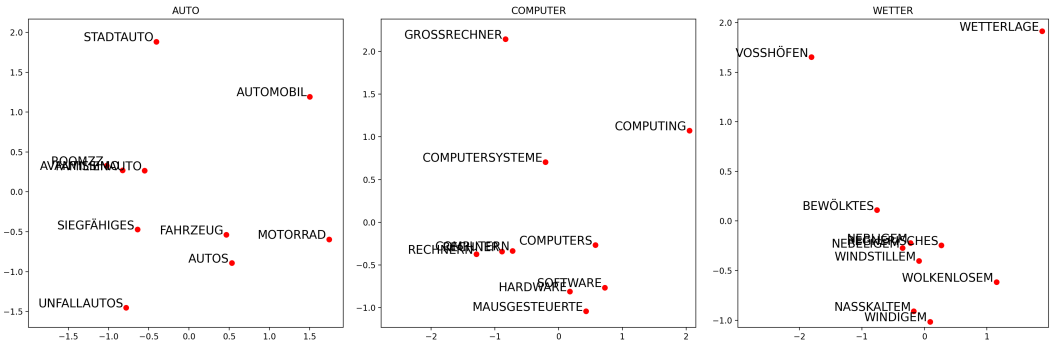

Figure 4.8: PCA visualizations for Word2Vec neighbors.

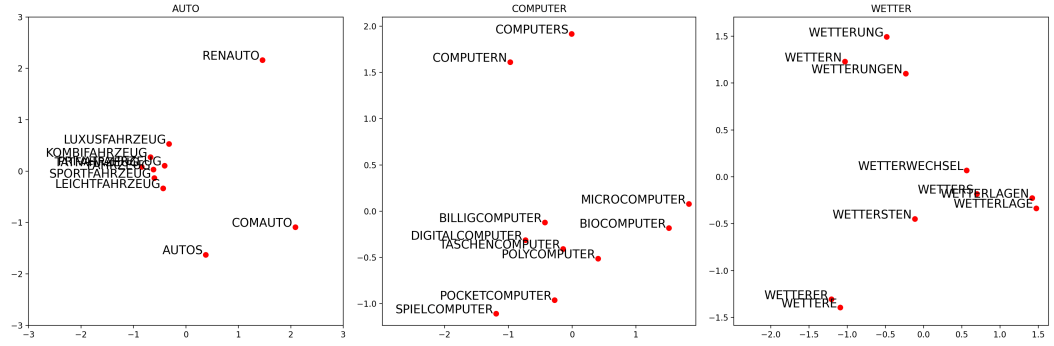

Figure 4.9: PCA visualizations for fastText neighbors.

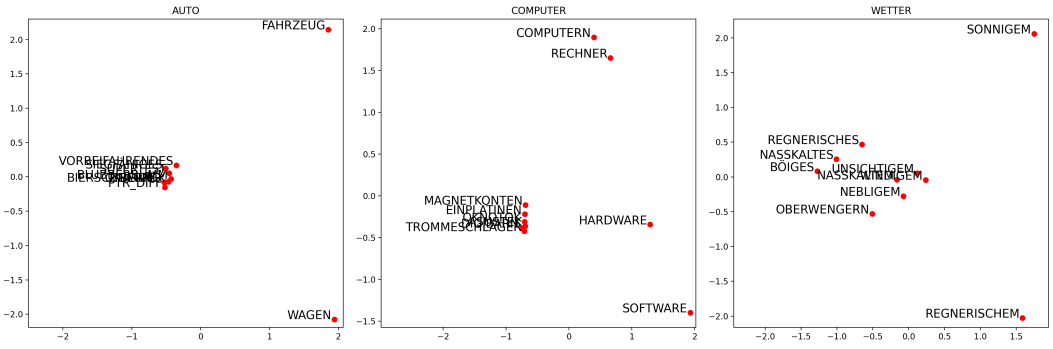

Figure 4.10: PCA visualizations for GloVe neighbors.

From the above visualizations, it can be postulated that Word2Vec and fastText select often synonyms as nearest neighbors while GloVe is more likely to select specifiers as nearest neighbors (e.g. adjectives which are often used to describe the queried word).

4.5 Generation of artwork embeddings

Algorithms like Word2Vec, Glove or fastText can not be directly applied to ARTigo artworks, because they are not just simple words from a corpus.

To get around this problem first a vector space from words (word embeddings) has to be generated and then a suitable vector for each artwork has to be calculated that will correspond to a perfect representation of the artwork among side all existing word vectors inside the vector space. Current artwork annotations can be used to retrieve vectors from the newly created word embeddings. After this the retrieved vectors will get converted to one vector, that represents the artwork, using mathematical operations and normalization.

The implementation consists of a python script with the following three steps:

- First annotations are fetched from PostgreSQL Database using a suitable query that only counts annotations that occur three or more times. The counting is used to avoid adding not sufficiently validated annotations to the process that were for example added by new users or by mistake.

- Second the program goes through every artwork and counts the occurrence of every unique annotation and the total amount of annotations for every artwork. Using that data the importance or weight of every annotation is calculated by using the simple term frequency formula, which essentially divides the count of every annotation occurrence by the total sum of annotations.

- Finally existing word embeddings for every annotation are fetched and dimensions are calculated by adding the retrieved vectors multiplied by their corresponding weight together. This step is also called normalization, which ensures that artworks with fewer annotations are just as equally important as artworks with more tags. Every resulting vector represents an artwork and all of them are written to a file.

4.6 Vector space search engine

Since vector space search engines are quite complex (see section 2.3 of this thesis), the use of a well-known existing solution, which is proven to work reliably, is more viable, than to create a new one from scratch. Many existing libraries are available for this task. Three of them are compared by speed and accuracy.

4.6.1 Comparison between Annoy, NGT, and NMSLIB

Using 100 distinct search keywords the time period starting from the request and ending with a reply from the search server, was measured using the python time module. The amount of nearest neighbors requested was 100 and to ensure accurate measurements, a delay of 1 second was added after each reply. The used word embeddings have 300 dimensions and are created with Word2Vec using a Wikipedia corpus.

NGT performed significantly worse than Annoy and NMSLIB, with an average response time of 40 milliseconds (using angular distance). Even using the, in theory, faster cosine distance, NGT was not able to come close to the others in terms of performance. With a measured worst-case delay of close to 100 milliseconds, it could result in a noticeable (by the user) delay in delivering search results and is therefore not an acceptable solution. The reason could be the graph-based indexing used by NGT internally and in that case, a lower requested amount of nearest neighbors would likely improve the performance.

NMSLIB performed almost as well as Annoy, with an average response time of 1.7 milliseconds. But, because of nonavailability was measured with cosine, instead of angular distance. A comparison of both NGT methods tested can be used as a performance reference between angular and cosine distances.

Finally, Annoy consistently outperformed the others with an average response time of just 1.5 milliseconds and is therefore the clear winner in terms of performance for the tested word embeddings. Time measurements in seconds are the following:

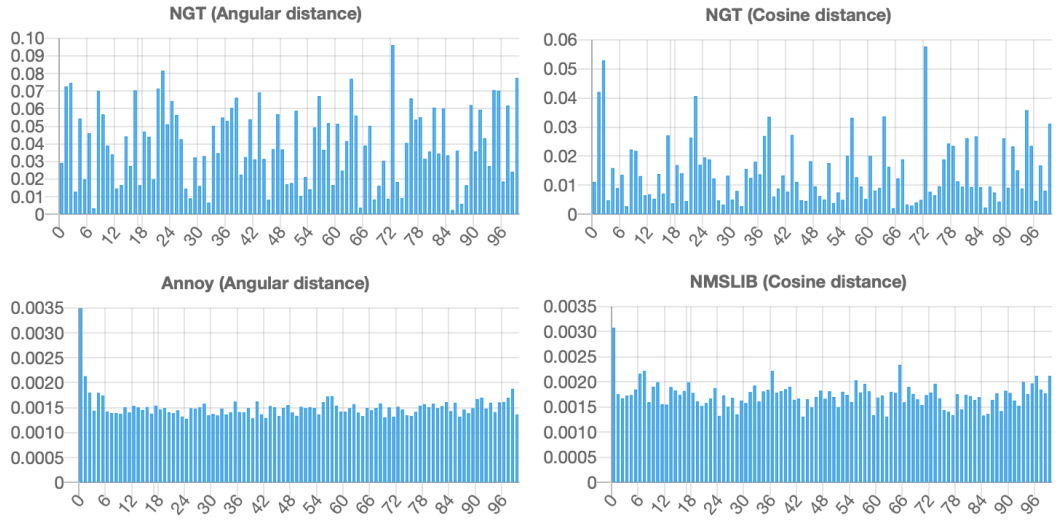

Figure 4.11: Vector search engine performance.

Additionally, Annoy has a function that can search for an existing vector in its index. By

creating an additional index for the word embeddings this function can then be used to find the necessary vector instead of loading the large word embeddings file into memory. This approach (by using a second index) lowered the system memory usage for Annoy from 8.94GB to just 1.96GB without a measurable performance difference. The reason behind this is the efficient storage method of Annoy's indexes.

The final implementation choice is Annoy, because it has the lowest response time, reduced the memory necessity, and can be implemented with Angular distance which in return should deliver the preferred search results. Thanks to a very good documentation Annoy was also easier to implement than NGT and NMSLIB.

4.6.2 Required files

Since the artwork identifies contained in the ARTigo database are not ordered, an artwork vocabulary has to be created that will map the saved artwork embeddings from section 4.5 to a real ARTigo artwork id's. It is required because the indexes of the vector space search engines do not support un-ordered identifiers or identifiers with a larger gap in between them. The artwork vocabulary is generated with a simple python script that loops through the existing artworks in the database and creates a new artwork-vocab file. When the server is running, this additional file is loaded into memory to ensure optimal performance.

4.7 Python search server

When the user searches for artworks a request with the search input is sent to the Node.js back-end and from there transmitted to the Python search server using a socket communication library called ZeroMQ, which was discussed in the back-end section (4.2) of this thesis.

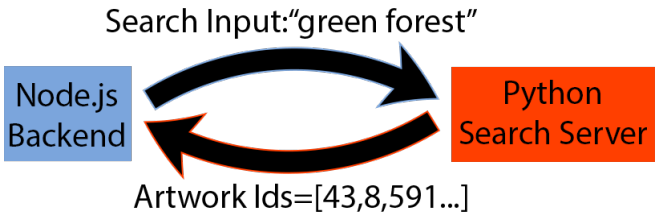

Figure 4.12: Connection between the Node.js back-end and the Python search server.

After the Python search server receives the request, it uses a vector space based search engine (see section 4.6 of this thesis) to locate the closest matching artworks and then sends their Identifiers back to the back-end.

A daemon-based approach (server running separately from back-end) was chosen because the search process requires additional files that have to be loaded into memory, before a search can be processed, which requires some time to load (usually a couple seconds). Therefore direct command-line calls (CLI) from the back-end are not an option. The required files are:

- Artwork embeddings index for the vector space search engine.
- The word embeddings file or only for Annoy: second index for word embeddings which lowers memory usage.

- Artwork vocabulary (as described in section 4.6.2 of this thesis).
- A data structure containing the word vocabulary among identifiers for each word. Used to retrieve a vector that corresponds to the search term more efficiently (discussed in section 4.7.2 of this thesis).

4.7.1 Requirements

The user has the ability to type in any word he wants in the search field and after the search server receives the user input, it needs to look up a corresponding word vector from the word embeddings file. But since this file is very large, just iterating over it until the word is found, is not an option, because in the worst-case scenario every word has to be compared and the performance impact would be significant.

To fix this problem, first a vocabulary file, containing all words, but without the vectors is created from the word embeddings file. Then a suitable data-structure is needed that can save all words from the vocabulary, with their indexes (which correspond to the line number of the word embeddings file, that contains the word) and has an efficient search function.

4.7.2 Data-structure for storing words

Data-structures that can fulfill these requirements are binary trees because they can drastically reduce the comparisons needed to find the index of a word. After the index has been found, the vector can be retrieved immediately. Depending on the vector space search engine, from an additional index, or if not possible - directly from the word embeddings file. Binary trees fall into two categories - balanced and unbalanced.

An example of a not balanced tree structure is the easy to implement and popular binary-search-tree (BST) and an example for a balanced tree is the AVL tree (named after its inventors Adelson-Velsky and Landis). An unbalanced tree can have any height, in the best-case $O(\log n)$, and in the worst-case $O(n)$ [40]. Balanced trees on the other hand have almost always the same height of right and left subtrees, which means that the height is $O(\log n)$, even in the worst-case [41]. Since the search time in any tree depends on its height, the balanced AVL trees are definitely better performing in terms of search time when compared to BST trees. For this reason, the implementation chosen is an AVL tree.

AVL tree improvement
The use of an AVL tree for searching through the whole word vocabulary for a requested word requires an average time 0.07 milliseconds while iterating through the whole word embeddings file can result in a worst-case time requirement of more than 100 milliseconds (if the last word in the vocabulary is the requested search term). Both measured with the python time library for taking the mean of 100 requests.

Pickle
To achieve faster start-up times of the search server, a method for saving and later loading the AVL tree from storage is needed. For python, the module Pickle, which is included in the standard library is suitable. The pickle module implements binary protocols for serializing and de-serializing a Python object structure. “Pickling” is the process of converting a Python object hierarchy into a byte stream, and “unpickling” is the inverse operation. Pickle greatly improved the start-up time of the Python search server from over a minute to just a couple of seconds.

Updating of existing artwork embeddings
ARTigo annotations are constantly updated using the GWAP games with new tags added to artworks every day. The search engine implemented in this bachelor thesis can be updated in a set period of time to persistently improve the search accuracy. For this, a Python script was created that loops over all artworks and then replaces the existing artwork embeddings. Implementation details were discussed in section 4.5 of this bachelor thesis.

4.7.3 Search result quality evaluation

Choosing words used for evaluation
To evaluate the quality we will look at results for search queries that have been **often**, **rarely** and **never** used as annotations. The following steps are required for the evaluation:

First, a simple SQL-Query over the ARTigo data-set is used with the two columns: annotation-word and count-of-times-annotated. The query also filters out words annotated less than 3 times (to avoid adding not suitable annotations that were for example added by new users or by mistake) and annotations for languages other than German. The results are 55792 records and the most common word was annotated 109314 times.

If we calculate the sum of our SQL-Query count-of-times-annotated column and then divide it by the total number of records (55792) we get the average amount, that a word has been used across all artworks, which is 161. Choosing words from the records, that have been used exactly 161 times across all artworks, should be a good enough estimation for often used words. For rarely used words, choosing any with an annotation-count of not more than 3 should be suitable.

Finally, for choosing never used words, since the goal of word embeddings is to capture the semantic meaning of words, it makes sense to look at currently existing annotations and choose similar words that are semantically related to the existing annotations but are actually never used as an annotation for ARTigo artworks.

Evaluation using test persons
The choice for a vector space search engine is hard to make with just using PCA or by searching for nearest neighbors for selected search queries. For this reason, a much better approach is to use test persons (voters) which will decide for a method, they find delivered the best artwork results for each query. For the preliminary evaluation, 10 participants and 10 pre-defined search queries are used. So in total 100 search results are examined by the test persons and evaluated. For Word2Vec and fastText the Skip-Gram and CBOW algorithms were trained with both 5 and 15 iterations. GloVe was tested with 15 and 25 iterations. All methods were trained using the same Wikipedia corpus and for every generated word embeddings file corresponding artwork-embeddings were created using the same algorithm which was specified in section 4.5 of this bachelor thesis.

According to the test participants, the use of 15 iterations for Word2Vec and fastText with both CBOW and Skip-Gram brought very poor results. Even for words that often appear as annotations like for example the German word “Baum” it delivered just a few matches. It could be due to overfitting. 25 GloVe iterations, on the other hand, resulted in more desired results, compared to 15 iterations but the observation was made, that GloVe delivered fewer matches for rare occurring words like for example the German word “Flugzeug”. Very good results were obtained by Word2Vec using 5 epochs, GloVe with 25 iterations, and fastText with 5 epochs. But mostly all voters choose for Word2Vec using Skip-Gram and 5 iterations because in their opinion it delivered a better connection between synonyms which turned out to be an important factor (see Figure 4.13). Therefore Word2Vec using Skip-Gram and 5 iterations is the final implementation choice.

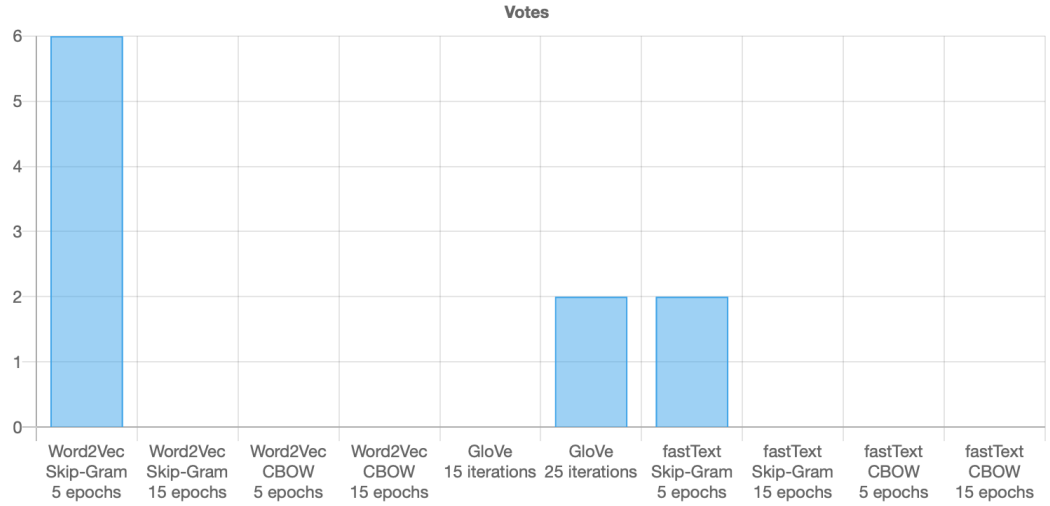

Figure 4.13: Amount of votes for vector space search engines.

For illustration purposes, we will look at a comparison between the best and worst-performing algorithms according to the test participants. On Figure 4.14 it can be seen that the best performing search engine (on the right side) returned more viable artworks.

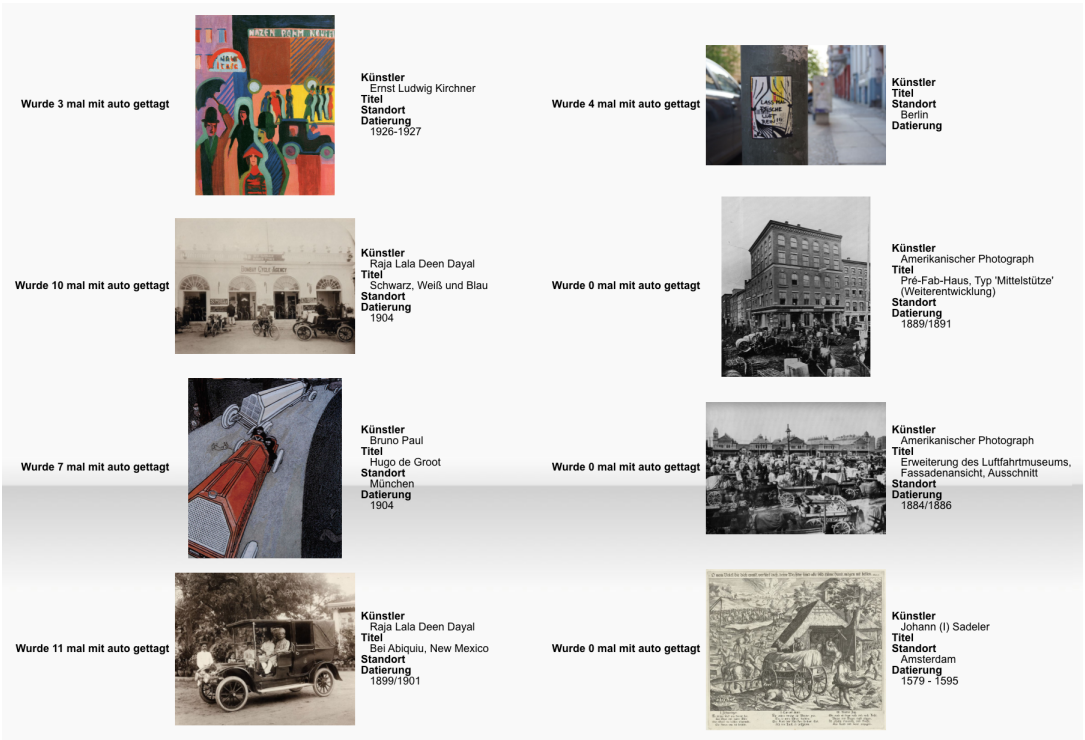

Figure 4.14: On the left side search results for Word2Vec using Skip-Gram with 5 iterations. On the right side search results for GloVe using 15 iterations. Both for the same query.

CHAPTER 5

Conclusion and Future Work

5.1 Conclusion

The search engine based on word embeddings implemented in this bachelor thesis definitely improved the existing search engine. Thanks to its large vocabulary size of 2.15 million words pretty much all words that come in mind are found in the word embeddings. Around 675 times more than all distinct annotations in the ARTigo data-set combined. Also, if a word embedding is found, the whole collection of artworks are always guaranteed to be available as results, because for every artwork all other artworks can be found with the nearest neighbors search. Performance is great with an average response time of just 1.5 milliseconds (see section 4.4 and 4.6.1 of this thesis) for fetching 100 artwork results.

Users experience a search engine with enriched results that seems to understand their request thanks to the semantic connection of words.

5.2 Future Work

5.2.1 Limiting factors

The work reported in this bachelor thesis depends on annotations of artworks so far provided which have to exist on the artworks because they are required for the algorithm that generates artwork embeddings. Without the annotations, it is currently not possible to generate artwork embeddings. Thus artworks without enough annotations cannot be provided with artwork embeddings, which is a problem in the development of a search engine. New methods for adding more annotations would certainly further improve the vector space based search engine implemented in this bachelor thesis.

5.2.2 In-depth analysis of word embedding generators

The generators used to create word embeddings in this bachelor thesis have many more options and configurations available that could not be all analyzed within the scope of this thesis. For the Wikipedia corpus used, Word2Vec's negative-sampling, GloVe's X-max

value, fastText's length of word-N-grams, and the content-window-size for all three methods, are examples of additional properties, that could affect word embedding accuracy. For this bachelor thesis only recommended or default values were used. These can be further in-depth examined.

5.2.3 Intelligent search interface with improved query language

The search interface implemented in this bachelor thesis can only search for one word at a time and is mostly for proof of concept. It could be further improved by adding the ability to search for more than one term. Additionally, support for custom weights could be added so that the importance of each searched term can be adjusted by the user. An ability to exclude artworks tagged with a specified word or an ability to specify which annotations must be included are more examples. Functionalities like these were implemented in existing search engines like for example elastic-search [6] or Apache Lucene/SOLR [35]. The query language supported by the Python search server implemented in this bachelor thesis could be extended by adding such additional functionalities which in return would deliver an improved search experience for the user.

5.2.4 Feedback loop

A feedback loop is a circuit that feeds back some of the output to the input of the same system. Feedback loops can be used to improve the Python search engine implemented in this bachelor thesis. Information about which artworks were clicked by the user for a given search query can be saved and later used to adjust the order of search results for the corresponding query. The idea is to enhance the search results order for the next queries with the same search terms by memorizing which results the user preferred.

Google uses a feedback loop technique to rank search results accordingly but no details about it are publicly available. The order of search results returned by Google is based, in part, on a priority rank system called "PageRank" [30]. Over the years Google has added many other secret criteria for determining the ranking of resulting pages. Exact implementation details are kept a secret to avoid difficulties created by scammers and help Google maintain an edge over its competitors globally [11].

Appendix

Before starting, the current ARTigo repository can be found on: `https://gitlab.pms.ifi.lmu.de/bognerm/artigo_ng`

Clone the project and go through the existing Readme file to set it up completely.

Obtaining word embeddings

Word embeddings can be obtained by either downloading existing pre-trained ones or by training a corpus.

Pre-trained word embeddings for the German language can be downloaded from `https://fasttext.cc/docs/en/crawl-vectors.html`.

For creating word embeddings from scratch, Wikipedia dumps with German articles can be downloaded from `https://dumps.wikimedia.org/backup-index.html`, by searching for "dewiki". After this the dump can be prepared using the python script `make_wiki_corpus.py`, which is included in the ARTigo repository, like this:
`python3 make_wiki_corpus.py <wikipedia_dump_file> <corpus_file>`

Training word embeddings

To build Word2Vec, GloVe and fastText, generally a modern Mac OS or Linux distribution and a compiler with good C++11 support is required (g++-4.7.2 or newer) or (clang-3.3 or newer).

Training with Word2Vec

The original implementation of Word2Vec from 2013, can be downloaded from the following GitHub Page: `https://github.com/dav/word2vec`.

Preparation:
After downloading, the program has to be compiled first by running make inside the src folder. This will generate word2vec executable located in the bin folder.

Recommended Parameters:
It is recommended to use a window size of 10 and 5 iterations. For a Wikipedia corpus, it is recommended to choose a size of 300 dimensions.

Additional Parameters:
For choosing between Skip-Gram and CBOW the parameter: -cbow (value) is used, 0 is for Skip-Gram and 1 is for CBOW.

To achieve maximum performance the number of threads have to match with the CPU threads available.

Start training:
From the bin folder execute the following command

```
./word2vec -train corpus-location -output word-embeddings-location
 -size 300 -window 10 -cbow 0 -threads 16 -iter 5
```

Training with GloVe

GloVe can be downloaded from the official GloVe Github repository: `https://github.com/stanfordnlp/GloVe`.

Recommended Parameters:
It is recommended to use 15 - 25 iterations with a window size of 15 and a X-Max value of 100. For a Wikipedia corpus, it is recommended to choose a size of 300 dimensions.

Start training:
To start training the included demo.sh script can be used, which first compiles GloVe using make, and then starts the training process.

Training with fastText

FastText can be downloaded from the official Facebook-Research GitHub repository: `https://github.com/facebookresearch/fastText/`.

Preparation:
After downloading, fastText has to be compiled first by running make inside the main folder. This will generate the fastText executable located in the same folder.

Recommended Parameters:
It is recommended to use 5 iterations and for a Wikipedia corpus it is recommended to choose a size of 300 dimensions.

Additional Parameters:
To select Skip-Gram choose the parameter: skipgram, and for CBOW the parameter: cbow

Start training:
From the main folder execute the following command

```
./fasttext skipgram -input corpus-location -output
 word-embeddings-location -dim 300 -epoch 5 -thread 16
```

Starting the Python server server

Python version 3 is required and all additional packages can be installed using `pip3`. To start the Python search server the following packages are needed:

- zeromq for communication with Node.js
- annoy for searching through the artwork embeddings

Requirements for first start-up:

- The package psycopg2 used for communication with the PostgreSQL Database.
- Open `zeromq_server.py` and specify the path of word embeddings file.
- Generate additional files by running the script `generate_required_files.py` once.
- Generate artwork index by running `build_artwork_index.py` once.

To start the server, simply run `zeromq_server.py`. If all required files, including the index, were successfully loaded, a message indicating the running status will appear.

To stop the server, simply terminate the python process.

5.2.5 Updating artwork embeddings

To update artwork embeddings (when new annotations are available in the database):

- Stop the python search server, if running.
- Run script `update_artwork_vectors.py` once.
- Start the python search server.

Bibliography

[1] Hervé Abdi and Lynne J Williams, *Principal component analysis*, Wiley interdisciplinary reviews: computational statistics **2** (2010), no. 4, 433–459.

[2] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim, *On the surprising behavior of distance metrics in high dimensional space*, International conference on database theory, Springer, 2001, pp. 420–434.

[3] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov, *Enriching word vectors with subword information*, Transactions of the Association for Computational Linguistics **5** (2017), 135–146.

[4] Jonghyun Choi, Hyunjong Cho, Jungsuk Kwac, and Larry S Davis, *Toward sparse coding on cosine distance*, 2014 22nd International Conference on Pattern Recognition, IEEE, 2014, pp. 4423–4428.

[5] Erik Bernhardsson (creator of Annoy), *Youtube talk on: Approximate nearest neighbors and vector models, introduction to annoy, https://www.youtube.com/watch?v=qkccylw0ehu*, Uploaded 13 October, 2015.

[6] Manda Sai Divya and Shiv Kumar Goyal, *Elasticsearch: An advanced and quick search technique to handle voluminous data*, Compusoft **2** (2013), no. 6, page 171–171.

[7] Readme.md document from official Facebook AI Research Github Repository Website, *https://github.com/facebookresearch/faiss*, Retrieved 10 July 2020.

[8] Readme.md document from official Node.js Github Repository Website, *https://github.com/nodejs/node*, Retrieved 4 September 2020.

[9] Official documentation Website of Gensim KeyedVectors model, *https://bit.ly/35wqkoc*, Retrieved 21 September 2020.

[10] Pedro Domingos, *A few useful things to know about machine learning*, Communications of the ACM **55** (2012), no. 10.

[11] WIRED Exclusive: How Google's Algorithm Rules the Web, *https://bit.ly/302m9yi*, Feb. 22, 2010.

[12] Introducing JSX Page from the React.js official Documentation Website, *https://reactjs.org/docs/introducing-jsx.html*, Retrieved 27 September 2020.

[13] React Without JSX Page from the React.js official Documentation Website, *https://reactjs.org/docs/react-without-jsx.html*, Retrieved 27 September 2020.

[14] Rendering Page from the React.js official Documentation Website, *https://reactjs.org/docs/rendering-elements.html*, Retrieved 27 September 2020.

[15] Is Node.js frontend or backend? Why Node Js Is Best Option for Back-end Development?, *https://www.technoexponent.com/blog/is-node-js-frontend-or-backend-why-node-js-is-best-option-for-back-end-development/*, Retrieved 28 September 2020.

[16] What is the Virtual DOM? Hey React, *https://medium.com/coffee-and-codes/hey-react-what-is-the-virtual-dom-466ec333bf9a*, Retrieved 27 September 2020.

[17] official JSON Website Introducing JSON, *https://www.json.org/json-en.html*, Retrieved 28 September 2020.

[18] Yury A. Malkov and D. A. Yashunin, *Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs*, CoRR **abs/1603.09320** (2016).

[19] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean, *Efficient estimation of word representations in vector space*, 2013.

[20] Mozilla Developer Network, *Introduction to the dom, https://mzl.la/2g3xmgw*, Retrieved 27 September 2020.

[21] Homepage of Ann-benchmarks Website, *http://ann-benchmarks.com/*, Retrieved 1 August 2020.

[22] Different Types of Distance Metrics used in Machine Learning, *https://bit.ly/3i58mjt*, Retrieved 5 September 2020.

[23] Chris Emmery (MSc Human Aspects of Information Technology), *Blog post about euclidean vs. cosine distance, https://cmry.github.io/notes/euclidean-v-cosine*, Tilburg University (March 25, 2017).

[24] Official Github Repository of NMSLIB, *https://github.com/nmslib/nmslib*, Retrieved 5 September 2020.

[25] Homepage of offical npm Website, *https://www.npmjs.com/*, Retrieved 28 September 2020.

[26] Homepage of React.js official Website, *https://reactjs.org/*, Retrieved 4 September 2020.

[27] Apache Lucene/Solr official Website, *https://lucene.apache.org/solr/*, Retrieved 8 Septeber 2020.

[28] ZeroMQ official Website, *https://zeromq.org/*, Retrieved 6 September 2020.

[29] A. Ojamaa and K. Düüna, *Assessing the security of node.js platform*, 2012 International Conference for Internet Technology and Secured Transactions, 2012, pp. 348–355.

[30] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, *The pagerank citation ranking: Bringing order to the web.*, Tech. report, Stanford InfoLab, 1999.

[31] Jeffrey Pennington, Richard Socher, and Christopher Manning, *GloVe: Global vectors for word representation*, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP) (Doha, Qatar), Association for Computational Linguistics, October 2014.

[32] the worlds largest web developer site React JSX Tutorial from w3schools.com, *https://bit.ly/2htcxeg*, Retrieved 27 September 2020.

[33] Andersen M.S. Ang Research Report: The LP Norm of Vector, *https://www.eee.hku.hk/ msang/vectorlpnorm.pdf*, Eletricial And Electornic Engineering (EEE) Department at the University of Hong Kong (February 25, 2013).

[34] Seyed Mahdi Rezaeinia, Ali Ghodsi, and Rouhollah Rahmani, *Improving the accuracy of pre-trained word embeddings for sentiment analysis*, arXiv preprint arXiv:1711.08609 (2017).

[35] David Smiley, Eric Pugh, Kranti Parisa, and Matt Mitchell, *Apache solr enterprise search server*, Packt Publishing Ltd, 2015.

[36] Kohei Sugawara, Hayato Kobayashi, and Masajiro Iwasaki, *On approximately searching for similar word embeddings*, Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 2265–2275.

[37] Liran Tal, *How much do we really know about how packages behave on the npm registry?, blog post on snyk.io, https://snyk.io/blog/how-much-do-we-really-know-about-how-packages-behave-on-the-npm-registry/*, April 22, 2019.

[38] Top 10 Reasons to use Node.js as a Back-end in the combination of React.js as front end, *https://medium.com/devtechtoday/full-stack-web-application-using-react-js-and-node-js-39c570b88d1e*, Retrieved 5 September 2020.

[39] Ellen M. Voorhees, *Natural language processing and information retrieval*, Information Extraction: Towards Scalable, Adaptable Systems (Berlin, Heidelberg), Springer-Verlag, 1999, p. 32–48.

[40] Wikipedia, *Binary search tree, https://bit.ly/3kt5yua*, Retrieved 20 September 2020.

[41] Wikipedia, *Avl tree, https://bit.ly/2gcfate*, Retrieved 21 September 2020.

[42] Google Code Archive word2vec, *https://code.google.com/archive/p/word2vec/*, Retrieved 9 July 2020.