TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TEPPO HJELT

NATURAL LANGUAGE PROCESSING TECHNIQUES FOR MEASURING WORD SIMILARITY

Master of Science Thesis

Examiner: prof. Tarmo Lipping
Examiner and topic approved on
26 November 2018

ABSTRACT

TEPPO HJELT: Natural language processing techniques for measuring word similarity
Tampere University of Technology
Master of Science Thesis, 67 pages
December 2018
Master's Degree Programme in Magament and Information Technology
Major: Software Engineering
Examiner: Professor Tarmo Lipping

Keywords: natural language processing, NLP, web scraping, artificial intelligence, AI, semantic similarity, semantic relatedness, word2vec, word embeddings, ontologies, language models

An artificial intelligence application considered in this thesis was harnessed to extract competencies from job descriptions and higher education curricula written in natural language. Using these extracted competencies, the application is able to visualize the skills supply of the schools and the skills demand of the labor market. However, to understand natural language, computer must learn to evaluate the relatedness between words. The aim of the thesis is to propose the best methods for open text data mining and measuring the semantic similarity and relatedness between words.

Different words can have similar meanings in natural language. The computer can learn the relatedness between words mainly by two different methods. We can construct an ontology from the studied domain, which models the concepts of the domain as well as the relations between them. The ontology can be considered as a directed graph. The nodes are the concepts of the domain and the edges between the nodes describe their relations. The semantic similarity between the concepts can be computed based on the distance and the strength of the relations between them.

The other way to measure the word relatedness is based on statistical language models. The model learns the similarity between words relying on their probability distribution in large corpora. The words appearing in similar contexts, i.e., surrounded by similar words, tend to have similar meanings. The words are often represented as continuous distributed word vectors, each dimension representing some feature of the word. The feature can be either semantic, syntactic or morphological. However, the feature is latent, and usually not under understandable to a human. If the angle between the word vectors in the feature space is small, the words share same features and hence are similar.

The study was conducted by reviewing available literature and implementing a web scraper for retrieving open text data from the web. The scraped data was fed into the AI application, which extracted the skills from the data and visualized the result in semantic maps.

TIIVISTELMÄ

TEPPO HJELT: Luonnollisen kielen käsittelyn menetelmät sanojen samankaltaisuuden mittaamisessa
Tampereen teknillinen yliopisto
Diplomityö, 67 sivua
Joulukuu 2018
Johtamisen ja tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Tarmo Lipping

Avainsanat: luonnollisen kielen käsittely, tekoäly, semanttinen samankaltaisuus, word2vec, web scraping, ontologiat, kielimallit

Diplomityössä käsiteltävän tekoälysovelluksen tehtävänä on louhia luonnollisella kielellä kirjoitettujen työpaikkailmoitusten ja korkeakoulujen opetussuunnitelmien tekstisisällöistä niissä esiintyvät kompetenssit. Louhittujen kompetenssien avulla pystytään visualisoimaan koulujen osaamistarjonta ja työmarkkinoiden osaamiskysyntä sekä näiden väliset yhtäläisyydet ja erot. Ymmärtääkseen luonnollista kieltä tietokoneen pitää pystyä arvioimaan eri sanojen samankaltaisuutta. Tämän työn tarkoituksena on etsiä ja esitellä parhaita keinoja avoimen tekstidatan louhimiseen ja sanojen semanttisen samankaltaisuuden mittaamiseen.

Luonnollisessa kielessä eri sanat voivat tarkoittaa samanlaisia asioita. Tietokoneen on mahdollista oppia sanojen samankaltaisuus pääsääntöisesti kahdella eri tavalla. Tarkasteltavasta sovellusalueesta voidaan muodostaa ontologia, joka mallintaa alueen käsitteet ja niiden väliset relaatiot. Ontologia voidaan ajatella suunnatuksi graafiksi, jonka solmut ovat sovellusalueen käsitteitä, ja niitä yhdistävät kaaret käsitteiden välisiä relaatioita. Käsitteiden välinen samankaltaisuus voidaan laskea niiden väliseen etäisyyteen ja niitä yhdistävien relaatioiden vahvuuteen perustuen.

Toinen sanojen semanttisen samakaltaisuuden laskennallinen toteamistapa perustuu tilastollisiin kielimalleihin. Malli oppii sanojen väliset yhtäläisyydet sen perusteella, minkälainen niiden todennäköisyysjakauma isoissa tekstikorpuksissa on. Sanat, jotka esiintyvät samassa kontekstissa eli samojen sanojen ympäröimänä, ovat yleensä merkitykseltään samankaltaisia. Moderneissa kielimalleissa sanat esitetään usein moniulotteisina sanavektoreina, joissa eri ulottuvuudet pyrkivät oppimaan jonkun piirteen sanasta. Piirre voi olla merkitykseen, syntaksiin tai sanan taivutukseen perustuva. Piirre on kuitenkin piilevä, emmekä yleensä pysty sitä päättelemään. Jos sanavektorien välinen kulma on moniulotteisessa piirreavaruudessa pieni, sanat jakavat samoja piirteitä ja ovat samankaltaisia.

Tutkimus on tehty tieteelliseen kirjallisuuteen perehtymällä ja toteuttamalla web scraper tekstidatan keräämiseen avoimilta verkkosivuilta. Web scraperin keräämä tekstidata syötettiin tekoälysovellukselle, joka etsi datasta osaamisia kuvaavat sanat ja visualisoi ne semanttisiksi kartoiksi.

PREFACE

As a former library guy, I have a funny affection for words. In Finnish language we have lots of them. I mean words, not funny affections. Some consider it a good thing, but I guess computers do not. Computers prefer unambiguous and precise expression. Sometimes I believe they may even hate Finnish and its dozens and dozens of different word suffixes. However, I want to help computers get along with their Finnish issues, as well as other language issues.

The last two courses of my studies, Deep Learning and Software Project, introduced me to the topic of natural language processing. Harri Ketamo and Antti Koivisto from Headai pushed me deeper by proposing the subject to this thesis. The effort is now completed, but it was just the beginning of some vague and exciting trip. During my first steps, I have learned that natural language processing is not an easy task. We speak with different words about the same subject or with same words about different subjects. Timo Honkela (2017) stated aptly in his AI-researcher's testament Rauhankone (Peace Machine): lack of common language could be an initiator to many crisis. Fortunately, mine was just a personal crisis of the thesis writer. And a temporary one.

I would like to thank my family for their patience; my fellow students and folks at Headai for their support; TUT and UCPori staff for all their help; and my examiner for keeping me on the right track.

Pori, 30.10.2018

Teppo Hjelt

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

3UAS, 3AMK	a co-operation partnership of the three Helsinki Metropolitan Universities of Applied Sciences: Laurea, Metropolia and Haaga-Helia
AI	Artificial Intelligence, capability of a computer to emulate intelligent human behavior
API	Application Programming Interface, abstracts the underlying implementation and exposes only objects needed to develop software
BOW	Bag-Of-Words, a technique representing a text as counts of its words, ignoring the word order
CBOW	Continuous-Bag-Of-Words, a word2vec implementation
CSS	Cascading Style Sheets, style sheet language used for describing the presentation of a document written in a markup language like HTML
DOM	Document Object Model, the browser creates a DOM of the loaded web page
GDPR	EU General Data Protection Regulation
HTML	Hypertext Markup Language for creating web pages
HTTP	Hypertext Transfer Protocol, foundation of data communication for the web
IC	Information Content, statistical quantity for word importance
IR	Information Retrieval
JSON	JavaScript Object Notation
KG	Knowledge Graph
LCS	Least Common Subsumer, the nearest common hypernym of concepts
LSA	Latent Semantic Analysis
MLP	Multilayer Perceptron, a class of feed forward neural network
n-gram	word sequence of n words
NLP	Natural Language Processing
OWL	W3C Web Ontology Language
POS	Part of speech
RDF	Resource Description Framework, a method for conceptual description of (web) information
RDFS	Resource Description Framework Schema
SOM	Self organizing map, a type of artificial neural network
spo	subject-predicat-object relation
TE	abbreviation for the Finnish phrase “työ ja elinkeino”, in English “employment and business”.
TF-IDF	Term Frequency – Inverse Document Frequency, statistical quantity for word importance
URL	Uniform Resource Locator, a web address
word2vec	word embedding, distributed vector representation for words
W3C	World Wide Web Consortium
WWW	World wide web
XML	Extensible Markup Language

1. INTRODUCTION

Schools offer education, which supplies students with skills. The labor market demands skills. However, do these skills match? Whether the skills supply meets the demand has remained a widely researched problem for ages (Scarpetta *et al.*, 2012). This study considers an AI-driven (artificial intelligence) approach to the problem. AI can be harnessed to read a huge amount of job advertisements and curriculum details, after which it figures out the skills offered by schools as well as skills needed in the labor market (Figure 1).

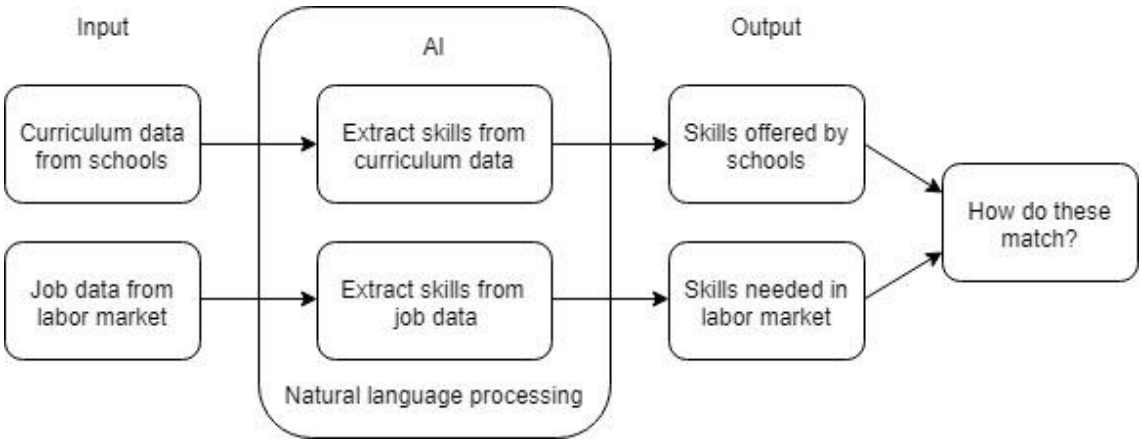

Figure 1. *Theme description of the thesis, the "big picture".*

AI outputs illustrative visual maps of skills supply and demand, from which we can find answers to various interesting and significant questions concerning curriculum development and related tasks, such as:

- What are the top skills needed in a particular field (e.g., computing or business) or in particular region (e.g., Helsinki region)?
- What are the top skills supplied by a particular school (the focus of the school)? What are the top skills supplied by a combination of schools (the focus of a partnership)?
- Do critical gaps exist? Does the school's supply of skills lack something desperately needed in the labor market? Or does it supply students with skills irrelevant in the labor market?

Why AI, this sounds like an easy task? It would be easy for a human, if there was a reasonable amount of data to handle. Unfortunately, there are far too many of these skills descriptions to go through manually. It would be easy for a machine, if the data were expressed in a machine understandable way, unambiguously without redundant noise. However, the skills descriptions are written by humans for humans. Moreover, they are written by various humans using various vocabulary with various words meaning the

same thing and the same word meaning various things, many words actually meaning nothing important. So much for the unambiguity.

We definitely need a machine to do the job and we need to make it understand natural language, especially semantic similarity and relatedness between different words and concepts. That is the focus of this thesis. Hence, we will pursue answers to the following research questions:

- What is the role of natural language processing in data driven curriculum development?
- How to extract semantics from words, which originally are just character sequences on a web page?
- How to measure semantic similarity and semantic relatedness between words?

During the process we will also follow the evolution of information from data to knowledge: at first we have only text strings on the web, then filtered text data in the database, then preprocessed and further filtered text to be processed with AI, then information in the form of semantic representation of words, and finally decision supporting knowledge to be exploited in curriculum development and related tasks. The whole process of data driven curriculum development is an interdisciplinary process beginning with Computer Science, ending somewhere in the middle of Educational and Management Sciences. However, the scope of this effort is limited completely to Computer Science. We will mostly deal with computational linguistics and natural language processing within the field of AI but will also study some text mining and information retrieval.

This work follows the guidelines of exploratory research (Stebbins, 2008). We search for the best designs for the semantic concept matching (in this case skills matching) between documents by proposing methods for online data collection, information extraction and semantic similarity measuring. Proposed methods are sometimes illustrated by simple examples. The study relies on reviewing available literature and introducing an AI implementation as a solution to the concept matching problem. This application is used as a black box. In other words, it is viewed only in terms of its input (text data) and output (skills) without knowledge of its internal details and algorithms (see Figure 1).

The organization of the thesis goes as follows. Chapter 2 introduces the reader with web scraping, a traditional data gathering technique used in this work. Programming a web scraper for retrieving and storing the curriculum data was the practical part of the thesis. Chapter 3 is the core of the work considering NLP techniques in word similarity measuring. The chapter begins by outlining a set of text preprocessing methods and the rest is dedicated to exploring ontology-based and corpus-based semantic similarity computing. Chapter 4 focuses on the practical implementation of the previously mentioned AI application. It covers the task specific data resources, i.e., curriculum data and job data, and demonstrates the semantic matching and visualization of skills extracted from them. The final chapter sums up the whole effort and reviews the results.

2. EXTRACTING DATA FROM THE WEB

For data driven decision making, like curriculum development, we need to analyze data. In recent years, due to exponentially increased computing power and likewise exponentially increased amount of available data, we tend to think all data analysis is about big data. By that somewhat abstract concept, big data, we usually mean data so vast and diverse it could not be perceived, acquired, managed and processed by traditional IT tools within tolerable time (Chen, Mao and Liu, 2014). Fortunately, there is still plenty of room for smaller data, too. Lots of important data are structured, or semi structured at least, and accessible by traditional means. That kind of data were harvested from the WWW, the World Wide Web (later referred to as the web), in the practical part of this thesis by a procedure called web scraping.

2.1 Web scraping

Web scraping is one of the oldest techniques for extracting web contents (Glez-Peña *et al.*, 2013). Later, information is extracted from the scraped data and further processed to become knowledge exploited in decision making. So, we will travel the classic path of the knowledge hierarchy: from data to information, from information to knowledge and finally wisdom (Henry, 1974; Rowley, 2007).

A Dictionary of Social Media describes web scraping as follows:

> *Extracting large amounts of data from an online source (often using an automated tool), especially where it is then reproduced somewhere else* (Chandler and Munday, 2016).

Even after this description, web scraping as a concept could appear somewhat vague to the reader. It is no wonder, quite similar concepts buzz around internet data related articles densely. For example, Technopedia (2013) sums the following terms up as synonyms: web scraping, web data extraction and web harvesting. They are just terms for various methods to collect information from the internet. There exist even more terms with related meanings in informal communication. We will use the term web scraping systematically throughout this thesis.

The main idea behind web scraping can be stated as follows: we program a software that accesses as many web sites as needed for the task, parses their contents to find and extract the data we need, and structures the data the in the most suitable way for us. Obviously there are three separate steps in the process: site access; HTML parsing and information extraction; and outputting the information (Glez-Peña *et al.*, 2013).

Web scraping could be easily confused with web crawling. The main difference between these two is that crawling just indexes the data it finds – like Google indexes (almost) all of the internet – whereas scraping *extracts* information from the data it finds. These two could be effectively combined, though: the crawler bot first indexes the data and then the scraper extracts information out of it (Massimino, 2016). The line between crawling and scraping is obscure. For example, when programming an automated web scraper, it is sometimes necessary to solve web addresses of popup-windows by concatenating strings. This is usually carried out by following some formula or pattern, which is more of a crawling kind of operation. Web crawling is not covered in further detail here, only to the extent it is embedded into web scraping. Curriculum data scraped in the practical part of this thesis was extracted from a few predetermined web sites, and in that sense did not involve web crawling.

2.2 Web scraping techniques

Sometimes, when extracting content from the web site, we might be able to take a shortcut compared to complete web scraping. That is the case, when we find a suitable data API (Application Programming Interface) to utilize. The data API provides a way to perform searches to the site and download content without retrieving and parsing the actual HTML code. The desired information is usually returned in JSON (or XML) format and can conveniently be inserted into database or whatever is chosen to be the storage for the information. Unfortunately, there still remain lots of domains lacking existing APIs or the APIs just do not give access to the desired data (Glez-Peña *et al.*, 2013). Hence, we need traditional web scraping.

Much of the current data driven research and development benefits a lot from well-designed web scraping. In fact, regardless of the web data related task, the ability to grab any online data, in any amount or format, storing and retrieving it in any suitable way, sounds like a very necessary skill for any data scientist (Mitchell, 2013). When designing web scraping, we create software that automatically extracts information from web pages originally designed for human use (Glez-Peña *et al.*, 2013). In other words, the scraper extracts the same information from the HTML code that human sees on the web page. The aim of web scraping is simply to mimick a human copying only relevant information from the web and pasting it to a more suitable repository. Of course, there are times, when human copy-paste-method easily beats automated scrapers, but that is only with extremely small amounts of data, never with AI-related tasks.

While programming a web scraper the best general programming techniques and good practices should be respected. Testing, proper use of methods and speeding up programs give the scraper more efficiency. For example, threading can speed up scraping noticeably. It could take a long time for web pages to respond, and multiple threads running simultaneously enable optimizing efficiency during these down times. Distributed computing could be even more powerful than threading. It also comes with another valuable

advantage. Using multiple cloud machines with distinct addresses to visit a web site might not be as alarming for a host trying to prevent unauthorized scraping as all requests coming from a single address. (Mitchell, 2013)

In order to fully understand what follows, the reader should first be introduced to the structure of an *HTML element*. HTML (Hyper Text Markup language) is the standard markup language for creating web pages and describing their structure. HTML elements, in turn, are the building blocks of the web pages, and hence crucial for web scraping. Elements are represented by tags, which label the content of the element. In the following code `h1` denotes a heading and `p` denotes a paragraph:

```
<h1>Main heading</h1>
</p>First paragraph…</p>
```

Elements can be enriched with additional attributes like `id` and `class`, which can be used to identify and classify elements. They are of great help in web scraping as we will later see.

```
<h1 id=”thisParticularHeader”>Main heading</h1>
<p class=”introductionParagraph”>First paragraph…</p>
```

Program 1. *HTML elements with attributes*

When the web browser shows the web page, the tags are not displayed, but only their content is. The interested reader can find more information about HTML and related topics on the w3schools web site[1].

2.2.1 Scraping static web pages

There exist two types of web pages, static web pages and dynamic web pages. Let us first examine the *static web page* parsing, because it is much simpler and more straightforward. Static page is usually coded in plain HTML. If we look at the *source code* of the page on our web browser, we will see the same information that we see on the visual page, this time only supplemented with HTML-tags and attributes. The source code file contains the whole visible text content of the web page and all references to its other visible parts, such as images.

The core of web scraping is to find the areas within a document, that contain the desired information for our purpose. According to Massimino (2016), scraping can be categorized into three general cases:

1. using embedded identifiers,
2. tree-based navigation,

[1] W3schools is the world’s largest web developers site at the time, https://www.w3schools.com

3. searching for contextual identifiers.

This is an eligible, progressively complex, order to explore web scraping, and we will follow it, too.

Using embedded identifiers is the ideal situation for scraping. This can be exploited if the site to be scraped contains well-marked-up HTML: elements are identified and classified systematically with proper attributes, like `id` and `class` (see Program 1). The web scraper can then easily fetch the important elements by these attributes.

Unfortunately, not all web pages are constructed systematically with decent attributes attached to all important elements. In this case, we might try the second way to approach scraping and use *tree-based navigation*, sometimes called HTML DOM parsing. Every time a web browser loads a web page, it creates a Document Object Model (DOM) of the page. DOM creates a tree of objects on the web page. DOM is not just HTML-related standard, though. Besides HTML DOM there exist XML DOM for XML-documents and Core DOM for all document types (W3schools, 2018). In our case the HTML DOM is sufficient, and it will later be referred to simply as DOM.

Let us take an example to clarify the concept of DOM. Human sees a web page on a browser screen as formatted text containing a heading, a paragraph and an unordered list (Figure 2).

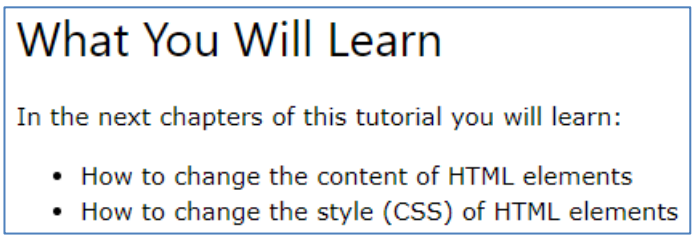

Figure 2. *Screenshot from a web page as human sees it (W3schools, 2018)*

The page written in HTML code contains the same information, but each section (in this case the row) of the page is coded as its own element. The visual content we see in Figure 2 is coded in HTML in Program 2.

```
<html>
      <head></head>
      <body>
            <h2>What You Will Learn</h2>
            <p>In the next chapters of this tutorial you will learn:</p>
            <ul>
                  <li>How to change the content of HTML elements</li>
                  <li>How to change the style (CSS) of HTML elements</li>
            </ul>
      </body>
</html>
```

Program 2. *HTML code implying a tree structure* (W3schools, 2018)

The elements in Program 2 clearly create a tree structure, the root being the `html` element. The whole DOM tree created from the elements of Program 2 is visualized in Figure 3.

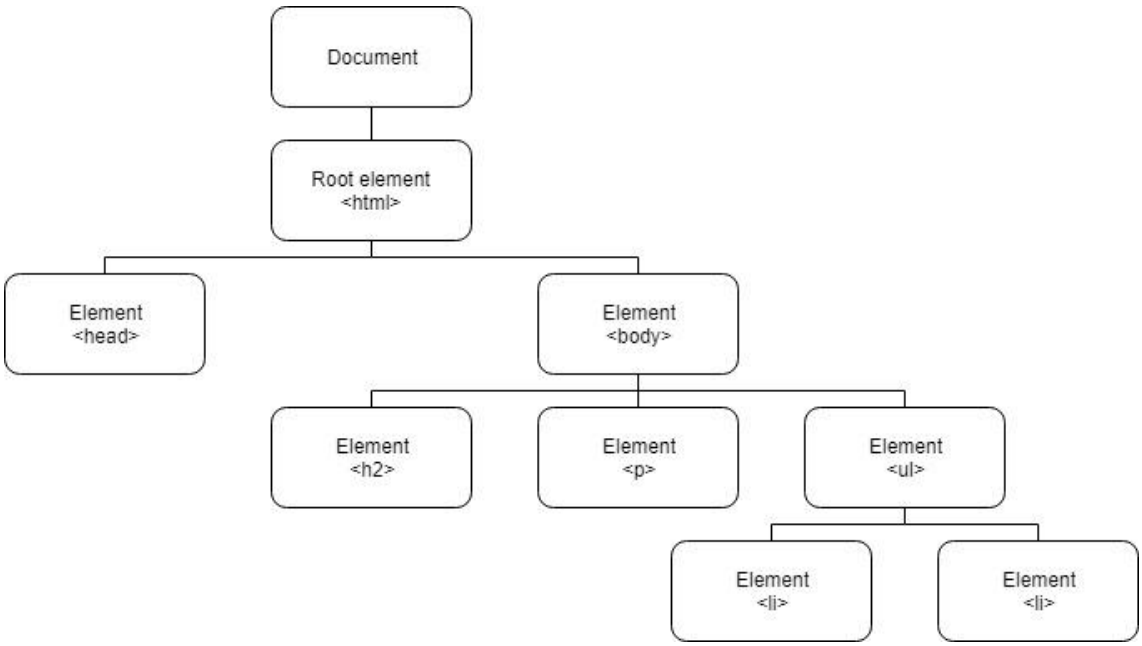

Figure 3. *HTML DOM tree created from an HTML code*

In tree-based navigation the web scraper traverses through the DOM tree and takes advantage of both the tree structure (parents, children, siblings) and the attributes of the HTML elements.

These two first scraping techniques are sufficient if the structure of the page is consistent and systematic. This is often the case considering pages with similar content provided by the same host, like curriculums from a certain school or jobs from a certain job site. However, sometimes pages with similar content could be coded by various persons or information on them could be extracted from various spreadsheets or database tables with various structures. This usually results in a web site having inconsistent tree structure without systematic identifiers (attributes). Inconsistent sites can be scraped by *searching for contextual identifiers*.

Contextual identifiers are usually some words on a web page giving a hint of the following words containing the desired information (Massimino, 2016). For example, text "*Language:*" on a course page reveals that the following word(s) might tell us the teaching language of the course. Using contextual identifiers usually leads to multitude of conditions: if condition 1, then solution 1, else if condition 2, then solution 2 and so on. This is not efficient and for the most part not complete either. There can always be a condition not taken into consideration, but we must settle for that. As Mitchell (2013) appropriately puts it, web scraping is often riding between the lines of what is intended and what is possible. Sometimes we have to write inelegant code to deal with inelegant code.

2.2.2 Scraping dynamic web pages

Considering *dynamic web pages*, the situation is more complicated. What distinguishes dynamic pages from static pages is that some of the information we see on the browser screen is not included in the source code file. Hence, it is not possible to be scraped by the techniques described above.

If we view the source HTML code of a dynamic web page on our browser, we will only find a part of the information visible on the actual web page. In fact, we will only see the static part. The dynamic parts of the web page are not included in the source code file but are *requested from the server* or from a database when the actual page is loaded. The page could also be all dynamic, in which case the source code does not include any of the visible information. The dynamic parts of the page are not written in HTML, but some server-side scripting language like PHP, Javascript or Java.

The information we see on the screen could change by clicking on a button, whilst the URL (Uniform Resource Locator, the web page address) of the page remains the same. Hence, the source code of the web page lacks the new information. With our browsers *Web developer tools*[2] we can try to solve how the new information is created. For example, if a popup-window appears, we might be able to solve how its contents are constructed, i.e., which external contents are called while the window is loaded. However, if we do not succeed in this, we can set up a virtual web browser to mimic human web browsing. This method will be considered in more detail in Chapter 2.3.

2.2.3 Storing the retrieved information

Storing the fetched information in a proper way and into a proper place is critical for further processing. There is no point in web scraping, unless the information found is not used in any way. Sometimes, if we have only little data, it is sufficient to store them into a spread sheet. Since web scraping is usually a part of some larger task, we need to store the data in a place where it is smoothly retrieved by other programs. *SQL database* is still

[2] Browser's Web developer tools are usually available by pressing F12 on the keyboard

a prevalent choice for storage especially with structured data, even though NoSQL databases are gaining ground among big data environment (Sadalage and Fowler, 2012; Mitchell, 2013; Reniers *et al.*, 2017). The same basic principles that concern database storage in general are applicable to web data, too. Mitchell (2013) and Massimino (2016) offer some advice while storing data scraped from the web. These could be capsulized as follows:

1. We must always sanitize the input, i.e. clean out the malicious code, while inserting into our database or when updating the database, because we never know what we might be picking up from the web. For example, we should use prepared statements.
2. In addition to the scraped information, it is always useful to store case specific metadata-information, such as author, language, revision history, comments, reviews.
3. We should also add a timestamp indicating when the information is retrieved from the web.

2.3 Web scraping tools

There are several frameworks and desktop-based environments to implement web scraping. They could be handy and perhaps more integrative, but a programmer with at least some experience might be most comfortable with customizing his/her own scraper using available language specific libraries and tools (Glez-Peña *et al.*, 2013). Massimino (2016) calls these tools semicustom software. Semicustom means that the programmer uses some well-documented open source library to handle low-level subroutines and supplements the code with his/her own implementations when needed. In addition to in-depth documentation available for the open source software, there is also continuous debate going on about the best practices for solving various kinds of problems with them in discussion boards and developer communities over the web, for example, Stack Overflow[3].

With a suitable combination of tools and libraries, the three-phase scraping process described in the previous chapter could usually be implemented in the following way (Mitchell, 2013):

1. retrieve the HTML code from a website,
2. parse it into an object,
3. isolate and process the desired data.

Although tools and libraries vary between programming languages, the main scraping principle remains the same.

To understand how the web scraping tools begin retrieving the HTML code, we need to know the very basics of *HTTP* (Hypertext Transfer Protocol). HTTP is a request-response protocol between local computer (client) and remote computer (server or host). The client

[3] The largest developer community at the time, https://stackoverflow.com

in our case is the web browser. An application that hosts the web site is the server. The browser submits a service request to the server, which returns the response to the client. For web scraping we need to retrieve the content of the web page. Hence, the browser sends a GET request for a certain URL (i.e., web address) and if successful HTTP communication occurs, the server responds by returning the content of the desired page to the browser. (W3Schools, 2018)

For example, if we want to retrieve the content of the web page from URL http://www.webaddress.com/folder/index.html, the browser sends a request including the following information:

```
GET /folder/index.html HTTP/1.1
Host: www.webaddress.com
```

Chosen programming language for the scraping task sets the options for available tools. The practical part of this thesis was programmed in Java, so the tools had to be Java compliant. By the help of the before-mentioned developer community the software chosen for the task was *Jsoup*, a Java *HTML parser*. Jsoup parses HTML code to the same DOM as modern browsers do and makes an object of the parsed document. The next simple example demonstrates how text information (course goals in this case) can be retrieved and extracted from the web page located at http://www.webaddress.com/folder/index.html using Jsoup:

```
// Get the document from URL location and save it as an object
Document doc = Jsoup.connect("http://www.webaddress.com/folder").get();
// Extract the elements with class=”goals” to an element list
Elements goals = doc.getElementsByClass(goals);
// Extract the text from each of these elements
for (Element goal: goals) {
    String goalText = goal.text();
    // Insert into database or list
    …
}
```

The previous example was naturally just an oversimplified case of data extraction. Jsoup enables the most complex DOM traversals and CSS (Cascading Style Sheets) selections for finding the relevant HTML elements from static web pages. Besides extracting the text contents of the elements as we did above, it is also possible to extract attributes (like URLs from links) and modify data with Jsoup (Hedley, 2009). Modifying data is irrelevant in information extraction, though.

With dynamic pages the HTML parsing tools are often insufficient. All the information on the dynamic pages is not included in the file retrieved by the parsing tool. The fetched page could include an element that has an attribute telling the browser to run a javascript code when the element is clicked on. Javascript code could load some additional information on the page or open a popup window, and we also need to retrieve that information. If we are able to resolve the URL of the loaded information by our web browser’s

developer tools, we have solved the tricky part of the problem. Then, we only need to feed this URL to our HTML parser tool, fetch the page and parse the information like before. However, sometimes we have to resolve hundreds or thousands or even more URLs of Javascript opened windows. In that case, if we are lucky, we can resolve the formula for creating these URLs. The formula could consist of some information of other elements, that could be extracted from the HTML source code. If we are not able to resolve the formula, we need a tool to execute JavaScript (or some other script language that loads the dynamic part of the page).

Web browser automation tools are of great help when it comes to dynamic web pages. Primarily these kinds of tools are for automating web applications for testing purposes, but they can conveniently be deployed in web scraping also. The idea is as follows. We set up a virtual web browser and code it to mimic human web browsing. The virtual browser can be coded to click on the elements on the web page and load their dynamic content on the browser screen. Now the information on the screen can be extracted with the automation tool in the same way that the HTML parsing tool did before. One such automation tool with support for the largest browser vendors is *Selenium*[4]. Selenium can be controlled by all common programming languages, including Java. The following example clarifies the performance of the Selenium tool. Let us have a simple HTML code including `h2` elements with `onclick` parameters, which load some new content on the web browser:

```
<h2 onclick=”loadGoals();”>What You Will Learn</h2>
<h2 onclick=”loadPrerequisites();”>Required Prerequisites</h2>
```

In this case, Selenium mimics a human web surfer in the following manner:

```
// Create a virtual web browser
WebDriver driver = new ChromeDriver();
// Get the document from URL location
driver.get("http://www.webaddress.com/folder");
// Extract the h2-elements to an element list
List<WebElement> headings = driver.findElements(By.tagName(("h2")));
// Click on each of these elements
for (WebElement heading : headings) {
    heading.click();
    // Extract information from the loaded content
} …
```

With Selenium we can also program our scraper to execute login procedures, search for specific information on the page and perform almost any web action like a human being would. Note, that this is only one of many ways to tackle scraping dynamic web pages. From the web, as well as literary sources concerning the subject, the interested reader can explore other approaches.

[4] SeleniumHQ browser automation, https://www.seleniumhq.org

In web scraping we are dealing with text data, so some additional text processing tools are usually valuable, too. For example, we might need something to help us manage, or rather avoid, regular expressions. Friedl (2002) describes regular expressions, also called regexp, as a powerful text data manipulating tool. With regexp it is possible to describe all kinds of text patterns, such as e-mail addresses following the pattern *user@domain.ending*, for example. We can tell the code what characters must be included in the pattern (in this case @ and .) and what characters are allowed to be between them. However, in more complex cases regexp can construct a very complicated and long expression and hence have a great potential for mistakes. In fact, Netscape's Jamie Zawinski's legendary quote from 1997 can be understood as an advice for inexperienced regexp users:

> *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.*

Usually programming languages have great tools to utilize in these situations, and they are definitely worth giving a try. During the practical part of this thesis, Apache Commons Lang[5] package, especially its StringUtils class, was in use and found very helpful in string manipulation.

2.4 Challenges considering web scraping

Web scraping for research, or some other general good, has usually good intentions. However, web scrapers are programmed for other purposes also, such as spamming. Therefore, the scraper, also a decent one, is in risk of being banned by the host. Web browsing is under continuous monitoring by some hosts to catch unauthorized crawlers and scrapers. That, in turn, results in problems complicating web scraping. Some of the problems are more challenging for web crawlers but are important to take into consideration also when programming web scrapers.

Data ownership comes into question, when scraping information from the web. Although the data is publicly accessible, it by no means signifies that its usage and ownership rights are transferred in the scraping process (Mitchell, 2013) . Hence, we always have to verify and follow the terms of use and copyright documents on the web sites that we scrape. For example, Twitter prohibits unauthorized web scraping in its Terms of Service:

> *"crawling the Services is permissible if done in accordance with the provisions of the robots.txt file, however, scraping the Services without the prior consent of Twitter is expressly prohibited"* (Twitter Inc., 2012)

[5] Apache Commons Lang provides extra methods to manipulate Java's core classes, http://commons.apache.org/proper/commons-lang

The `robots.txt` file tells the web crawler/scraper, which parts of the site are not allowed to traverse. It is placed in the top-level directory of the web site. A simple `robots.txt` file could tell all the robots to avoid visiting the `tmp`-folder of the site in the following manner:

```
User-agent: *
Disallow: /tmp/
```

`Robots.txt` uses the *Robots exclusion standard*[6], which is unofficial and that way legally not as binding as Terms of Service. It makes abiding `robots.txt` an ethical choice for the programmer (Mitchell, 2013; Kimmons, 2017).

In case no terms of use considering web scraping are provided, Massimino (2016) strongly recommends to contact the host to resolve the matter. However, Mitchell (2013) points out that there is no fundamental difference between accessing information on a web page via a browser and via an automated script like web scraper. Nonetheless, many sites will check to see if the visitor is actually a browser before sending the data. These sites check the HTTP header information we are sending with our every request. So, sometimes we must modify this HTTP header information (especially user-agent) on our scraper's request to make it look more like an occasional visitor on a web page:

```
"User-Agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/67.0.3396.87 Safari/537.36"
```

HTTP header information can be controlled with web scraping tools and libraries like previously mentioned Jsoup and Selenium.

Hosts might have implemented spider traps, i.e., endless loops of references generating no content, to trap the unauthorized crawlers (Massimino, 2016). These traps could also be generated unintentionally within a host's content generating process. As such, they annoy authorized web scraping as well. Instead, intentional spider traps could be avoidable. Although, being an important part of web scraping, creating undercover scrapers in more depth is outside the scope of this thesis. An interested reader is recommended to explore Mitchell (2013) or Massimino (2016) for further information on the subject.

[6] Robots exclusion standard: https://en.wikipedia.org/wiki/Robots_exclusion_standard

3. NATURAL LANGUAGE PROCESSING IN SEMANTIC SIMILARITY MEASURING

The information scraped and stored from the web contains just word strings, which are symbols without meaning. Is the word *car* more similar to the word *lorry* than to the word *cat*? The modern techniques of natural language processing help us measure the similarity between words and give them meaning.

3.1 Natural language processing or text mining?

The reader has probably come across terms such as *natural language processing* (NLP), *text mining* and text analytics, and wondered about the similarities as well as differences between them. The terms are all about analyzing text data, but are not synonyms, certainly not all of them. Oxford Reference, which brings together two million digitized entries across Oxford University Press's dictionaries and encyclopedias, gives us the following definitions:

> *"Text mining (text analytics): A form of data mining, which involves identifying and analysing patterns within a text and uses techniques drawn from natural language processing, machine learning, and statistics… Analyses that can be done include sentiment analysis, concept extraction, entity relation modelling, and text abstraction or summarization."* (Elliot *et al.*, 2016).

> *"Natural language processing: abbr.: NLP; the computational analysis and interpretation of human language. NLP is used in software that provides automatic translations of text from one language to another, in robotic systems that use human-language-type commands, and in text-mining tools (e.g. to provide summaries or abstracts of large volumes of text)."* (Cammack *et al.*, 2006)

From this we can conclude, that text mining and text analytics can safely be used as synonyms. In turn, NLP can be understood as a method used in text mining applications. NLP is the tool that helps the computer understand human language and text mining is the tool that extracts information from it. The target of both together is to produce applications that have human like comprehension of text, which allows them help humans make decisions based on a massive amount of text data. The key thing is to extract interesting and non-trivial patterns or knowledge from text documents. Without NLP and text mining this would not be possible or would at least be extremely difficult. It would take too much time, even if it was possible for a human to read all the text data involved.

From the point of view of this thesis it is not necessary to draw a strict line between these terms. In conclusion we can say that most of this chapter deals with NLP techniques in text mining.

3.2 Preprocessing of text data

Although job and curriculum data scraped from the web might not be as messy as in some other NLP tasks, it is not ready for further processing as it is. Around the web and in publications several frameworks or approaches for text *preprocessing* are introduced (Uysal and Gunal, 2014; Wang, Liu and McDonald, 2014; Elakiya and Rajkumar, 2017; Mayo, 2017). They give decent guidelines for designing text preprocessing, but choosing the best tactic always remains a task-specific, as well as a language-specific, problem. Preprocessing texts written in morphologically rich and highly inflected languages is a trade-off between computation time and thoroughness.

Preprocessing of text for NLP and text mining consists of procedures such as tokenization, removal of stop words (common meaningless words) and punctuation marks, stemming, lemmatization, part of speech (POS) tagging and substituting numbers with words. Principally, these steps reduce the amount of the words in the vocabulary by getting rid of redundant and duplicate information, hence making the final knowledge extraction more successful. We will group the preprocessing steps into three high-level concepts for simplicity: tokenization; text cleaning and substitution; and text normalization. In any case, no matter how we approach the subject, the steps will always remain partly overlapping. We are going to focus on the steps important with modern NLP techniques and morphologically rich languages, like Finnish.

3.2.1 Tokenization

Text *tokenization* is a fundamental preprocessing step for almost any text analyzing scenario and usually starts the whole preprocessing. According to Uysal and Gunal (2014) tokenization is a form of text segmentation, that splits larger text into smaller pieces, namely *tokens*. These tokens can be words, phrases or other meaningful text parts. Typically, word tokenization is performed considering only alphabetic or alphanumeric characters separated by space. Tokenizing is simpler for languages, where space is used as the word delimiter, such as Finnish. Rehman et al. (2013) study tokenization within languages with more complicated space usage, like some Asian languages and hand-written languages, but they are naturally beyond the scope of this thesis.

At first glance, tokenization might seem like a simple process: sentences are split by punctuation "." and words are split by whitespace " ". However, there are complex structures in written language, like the sentence *'"Mr. Holder is a Ph.D., and he's walking a full-time student Mr. Hill's dog", clarified Ms. Lea, when Mr. Powell wanted to know,*

what's going on.' The linear approach does not work anymore. There are numerous techniques available trying to tackle the tokenization problems for the languages of the world, for example, rule based, statistical, fuzzy, lexical and feature based tokenization (Rehman *et al.*, 2013; Mayo, 2017). Partially exploiting these techniques, the programming languages have their own tokenizer libraries. Still, reaching an eligible result often demands manual contribution from the programmer.

3.2.2 Text cleaning and substitution

Merriam Webster dictionary (2018) defines *noise* as "*irrelevant or meaningless data or output occurring along desired information*". As Xiong et al. (2006) emphasize, noise hinders most types of data mining. That makes data cleaning one of the most important phases in the knowledge discovery process. Considering text data, the focus is on cleaning the data from useless words or characters, that carry little or no semantic meaning and in the worst case, mess up the whole knowledge extraction. These irrelevant words and characters include *stop words*, symbols, emoticons (are they sarcasm or not?), mathematical equations and extra punctuation marks (Wang, Liu and McDonald, 2014; Alam and Yao, 2018). However, we usually want to keep sentence ending characters "." for AI tasks. There is no universal list of stop words. They usually consist of conjunctions, prepositions, articles and other meaningless words. They also depend on the given purpose. To give a simple example, stop words in English include words like *the, is, at, which, on* and in Finnish *ja, tuo, se, mikä, on*. Numerous stop word lists for numerous languages can be found from the web, but they are never complete and must usually be customized to meet the needs of the given task.

By the time of writing the thesis, we have just passed the GDPR (General Data Protection Regulation) enforcement day, May 25, 2018. According to European Comission (2018) GDPR was designed to protect and empower all EU citizens' data privacy. Now people have more control over their personal data. This reshapes the organizations' approach to data privacy in a way that no redundant personal data should be saved. This must be taken into consideration even more carefully than before and leads to data anonymization (or pseudonymization) before storing text data. There are several methods to perform anonymization, and they can be explored, for example, in Salas et al. (2018). At its simplest, anonymization can be done via substituting all personal data with xxx or "John Doe"-like data referencing to a hypothetical "everyman".

If numbers are considered important for the knowledge extraction process, they should be converted to textual representations. If numbers carry no relevant information, they are among the meaningless data to be removed. Also, rare terms are often removed, if they have no contribution to the knowledge extraction.

3.2.3 Text normalization, stemming and lemmatization

Above discussed text cleaning is sometimes considered a part of *text normalization*. Text normalization continues converting text into a format that enables more efficient knowledge extraction. Common techniques include converting all characters to lowercase and lemmatizing or stemming the text. The goal of normalization is to treat all forms of the word as one, making all text equal (Wang, Liu and McDonald, 2014). After normalization, the words *car*, *cars*, *Car* and *Cars* will all be treated as the word *car*, which becomes the representative for all its forms. Converting text to lowercase is very straightforward but stemming and lemmatization need to be studied more closely.

Finding the representative for all the inflections of the word is not always as simple as in the previous example of cleaning the plural suffix *s* from the noun *car.* English words *democracy*, *democratic* and *democratization* also have similar meanings and hence need to have a common form, or at least fewer than the original three forms. English is a weakly inflected language, and hence, the words do not have so many inflections. Instead, in highly inflected languages, such as Finnish, words usually have dozens of inflections. Stemming and lemmatization try to reduce this variation and find a common base form to present all the inflections. However, they use completely different methods to reach that goal.

Stemming is simpler and faster and makes the job usually by crudely chopping the suffix and keeping the *stem* of the word (Manning, Raghavan and Schutze, 2009). For example, the stem of the word *hitting* is *hit*. The most common algorithm for stemming in English is Porter's (1980) algorithm for suffix stripping. Later Porter extended his work to cover a variety of other languages, including Finnish in 2002. Porter described these stemmers in a high-level programming language called *Snowball* (Porter, 2014). Snowball stemmers are often used for stemming in modern NLP libraries of various programming languages.

However, stemming is not an easy task in highly inflected languages. Let us take the Finnish translation of the word *hitting* as an example. *Hitting* is *lyövä* in Finnish. It is an inflection of the word *lyödä*. Other inflections of the lemma *lyödä* include *lyön* (*I hit*, present tense), *löin* (*I hit*, past tense), *lyöt* (*you hit*, present tense), *löit* (*you hit*, past tense) and dozens of others. Only the character *l* in the beginning remains the same and the subsequent characters differ. So, stemming by chopping the varying suffix would not be very convenient in this case. In fact, Finnish words could have two stems, which makes the issue even more complicated. A lot of research has been done on highly inflected language stemming (Kettunen, Kunttu and Järvelin, 2005; Kettunen, 2006; Saharia *et al.*, 2013; Brychcín and Konopík, 2015; Dadashkarimi *et al.*, 2016).

The other normalization method, *lemmatization*, uses different methods and usually reaches a different result, too. The goal of lemmatizing is to find the *lemma*, the basic

form of the word. The basic form is the uninflected form of a word used as a dictionary entry. All nouns are lemmatized as singular lemmas. For example, *cars* will become *car*.

Lemmatization among highly inflected languages is even more complicated than stemming. The format of each word is dependent on the position and grammar connection with surrounding words and inflectional rules are spiced up with several exceptions. As Gallay and Simko (2016) point out, often the only option is to use an approach exploiting a dictionary of word-lemma mappings. The problem with this is that the dictionaries traditionally need to be created manually first determining the part of speech (POS) of the word and then applying different stemming rules for each POS. According to the authors, the creation of word-lemma dictionaries can be done semi-automatically using large corpora annotations. Still, neologisms (newly coined words) and new domain specific terms will evolve all the time. In any case, a human author or at least a human supervisor is required. This makes creating and maintaining dictionaries for highly inflectional languages a demanding task, if at all feasible. However, this kind of normalization, that takes the word dependencies into consideration, could be extremely helpful with unambiguous words, i.e., words with more than one meaning. If we know the POS of the word, we can distinguish between words *abstract* (adjective) and *abstract* (noun).

There has been an interesting opening in automatic word lemmatization recently by Gallay and Simko (2016) themselves. Their approach utilizes vector space word models, so called word embeddings. It is based on an assumption that word embeddings encode also morphological regularities in addition to syntactic and semantic ones. We will discuss word embeddings in in more detail in Chapter 3.5.4 and will get back to morphological regularity encoding as well.

Stemming and lemmatization do not differ that much in English, which is a weakly inflected language. Stemmed words often are lemmas, too. The situation is rather diverse in morphologically rich, highly inflected, languages. Korenius et al. (2004) studied the two methods in Finnish text clustering, concluding that lemmatization outperformed stemming in this setting. However, we have discussed the difficulties with lemmatization. Hence, the choice will remain task specific and a tradeoff between computation time and accuracy. If we prefer efficient real time applications, we must be ready for compromises.

3.3 Semantic similarity between words

The sentences "*I own a cat*" and "*I have a kitten*" obviously have very similar meanings. Still, they have only two words in common, *I* and *a*. If the text is thoroughly preprocessed, these common words are probably cleaned out. Based on this information the computer has no chance to capture the similarity between the sentences. We need to find a way to get the computer to find out that the words *own* and *have* hold similar meanings, as well as the words *cat* and *kitten*.

Semantic likeness between terms acts as a fundamental principle by which we organize and classify objects (Goldstone, 1994). Hence, to outperform human in this object classifying task, AI also must learn the semantic likeness between words. Therefore, computation of word likeness has become a popular research problem in AI and NLP fields tackling classification related problems such as word sense disambiguation, synonym detection, thesaurus generation, semantic text similarity, machine translation, information extraction and sentiment analysis (Curran, 2002; Patwardhan, Banerjee and Pedersen, 2003; Mihalcea, Corley and Strapparava, 2006; Chen, Lin and Chu, 2011; Cambria *et al.*, 2013; Li *et al.*, 2017).

It is important to understand, that there are different kinds of semantic likeness. In *semantic similarity* two terms share some aspects of their meaning. For example, *cats* and *dogs* are alike to the extent they are both mammals (or even more specifically pets). If the two words are synonyms, like *child* and *kid*, their semantic similarity is very high. *Semantic relatedness* refers to non-taxonomic, more general type of likeness, as in *car* and *wheel*, or *legs* and *trousers*. In fact, relatedness is what computational applications typically require, not similarity (Budanitsky and Hirst, 2001; Sánchez *et al.*, 2012). However, semantic similarity is more useful when applications need to capture the hierarchical relations between concepts, such as concept expansion (Dragoni, Da Costa Pereira and Tettamanzi, 2010).

The methods surveying computational semantic likeness between words can be divided into two categories: corpus-based methods and knowledge-based methods (Mihalcea, Corley and Strapparava, 2006). Both measure the likeness between words by gauging their distance from each other in their unique way.

3.4 Knowledge-based semantic similarity methods

According to Guarino, Oberle and Staab (2009), *knowledge-based methods* usually identify semantic similarity between two words by the help of artificial semantic resources, like ontologies. The roots of *Ontology* as a discipline (not yet being called Ontology, though) dealing with the structure of reality date back in the times of Aristotle. However, the prevalent use of the term from Computer Science's perspective is *an ontology*, a countable artifact, that formally models the structure of a system. In this thesis, by ontology we will always refer to the latter. The definition to an ontology was originally given by Gruber (1993), later refined by Borsts (1997) and finally these two merged by Studer (1998): *"An ontology is a formal explicit specification of a shared conceptualization"*.

While discussing knowledge extraction and semantic similarity measures, we will encounter several other terms related to ontology such as taxonomy, thesaurus, knowledge base, knowledge graph and semantic graph. From the perspective of semantic likeness, the key point is that knowledge can be presented, and hence traversed, as a labeled, *directed graph*. Distinguishing the terms from each other is not necessary. The reader can

consider all above listed representations as ontologies. In the following we will build a semantic graph beginning from a simple taxonomy and expanding our representation gradually.

For the most part, ontologies can be expressed as graphs (Harispe *et al.*, 2017). Figure 4 shows a backbone of an ontology, a *taxonomy*, as a directed graph. Taxonomy represents the hierarchy of some domain, forming a directed tree of classes and their subclasses. Hereby, the taxonomy classifies concepts. From the taxonomy in Figure 4 we can learn, that dogs and cats are mammals, which are animals, which are things. Similarly, cars and bikes are vehicles, which are objects, which are also things. Taxonomy has a taxonomic scheme which defines the properties considered to distinguish classes.

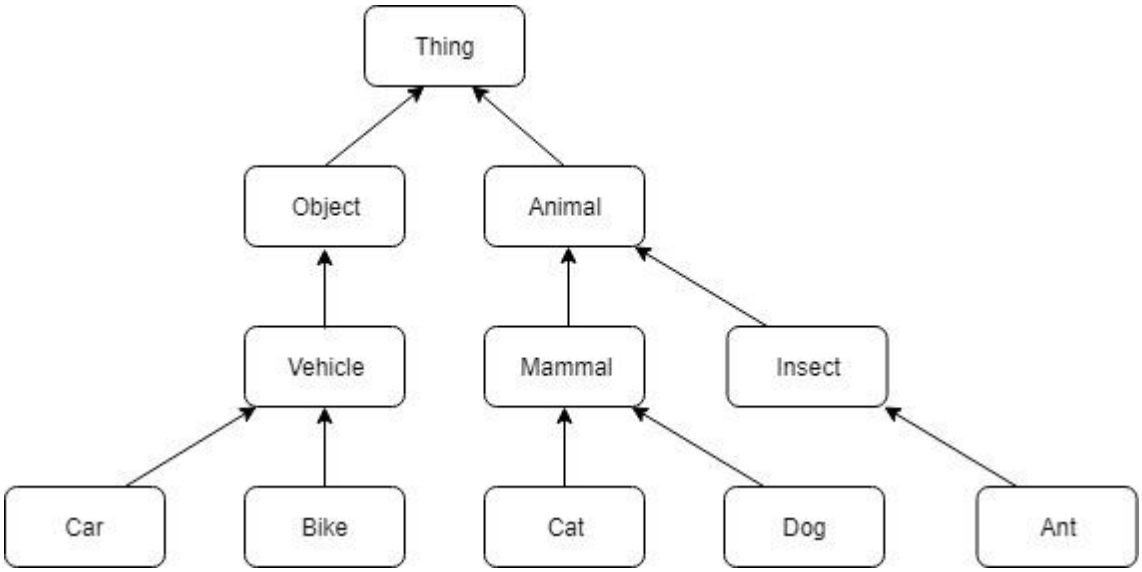

Figure 4. *Taxonomy as a directed graph*

According to Blumauer (2014), taxonomy becomes a *thesaurus*, when it is supplemented with non-hierarchical relations between concepts, such as synonyms or other related terms. Because of the non-hierarchical relations, thesaurus cannot be visualized as a tree, but as a more complex graph (network). However, thesauri are not competent enough to present the complexity of the whole knowledge of the world. Full ontologies supply us with more sophisticated relations.

All advanced approaches representing knowledge, like ontologies, share common components (Harispe *et al.*, 2017):

- *Concepts* (*classes*) are sets of things sharing common properties, e.g., *Cat.*
- *Instances* (*individuals*) are members of classes, e.g, *Tigger* (an instance of the class *Cat*).
- *Predicates* are types of relationships between instances or classes, which carry specific semantics, e.g., *subClassOf* (Car is a *subClassOf* Vehicle).
- *Relationships* (*relations*) are concrete links between classes and instances. Relationships form subject-predicate-object (*spo*) statements, which also carry specific

semantics, e.g., Tigger *isA* Cat, Tigger *isSiblingTo* Missy, Tigger *isBiggerThan* Missy, Cat *has* Fur

- *Attributes* are *properties* of instances (and classes), e.g., Tigger *hasName* "Tigger".
- *Axioms* are statements that say something about classes, instances, attributes, predicates or relationships, and comprise the overall theory the ontology describes, e.g., *Any Cat has fur, Any Cat has exactly 4 legs*.

Utilizing these common components introduced above, the taxonomy in Figure 4 can be extended to serve as an ontology, as perceived in Figure 5.

The ontology can be represented as nodes linked to their classes by simple spo-statements, hence forming a *semantic graph* representing semantic relationships between concepts (and instances). According to Harispe et al. (2017), all relationships which link the nodes of the graph, carry unambiguous and controlled semantics. There are two kinds of relationships in Figure 5: *hierarchical* relationships (indicated with blue text, like *isA*), that link subclasses (or instances) to their classes; and *non-hierarchical* relationships (indicated with green text, like *has*), that link classes (or instances) to their properties or instances to their data values. The semantic graph has been supplemented by three instances: *Tigger* and *Missy* of class *Cat* and *Lassie* of class *Dog* indicated with blue text boxes. They all have names as attributes, with values indicated with yellowish text boxes.

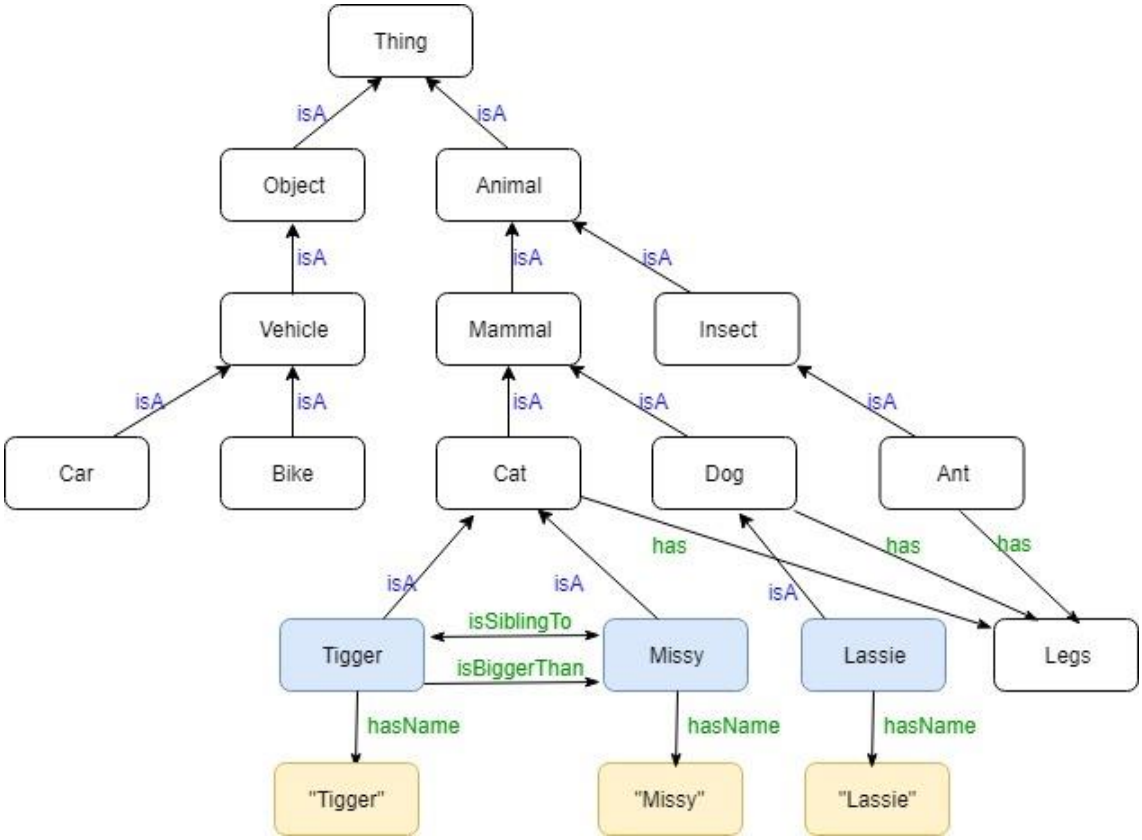

Figure 5. *A simple ontology extended from the taxonomy in Figure 4.*

From Figure 5 we can extract spo-statements such as *Car isA Vehicle*, *Tigger isBigger-Than Missy*, *Ant has Legs*. These spo-triples can be represented, for example, by RDF[7] (Resource Description Framework) and its extensions RDFS[8] (Resource Description Framework Schema) or OWL[9] (Web Ontology Language), which are all W3C (World Wide Web Consortium) standards. The interested reader is instructed to explore more on these vocabularies on W3C website as indicated at the footnote.

Spo-relationships in semantic graphs are utilized to define algorithms and characterize paths in the graph (Harispe *et al.*, 2017). Hence, they contribute to measuring the semantic likeness between the concepts. Semantic similarity relies on the hierarchical relations (like *isA*) between concepts and semantic relatedness accepts also non-hierarchical relations. Utilizing this information, we can conclude that *cat* and *dog* are semantically similar because they share the same direct hypernym *mammal*. Semantic similarity between *ant* and *cat* is not as big because their common hypernym *animal* is further away. *Ant* and *cat* are semantically related also because they both have legs. We can intuitively deduce, that semantic similarity metrics is proportional to the length of the path connecting the concepts. Several classical methods are introduced in the literature to transform our intuition into computable metrics.

3.4.1 Path-based methods

In what follows, we will moderately exploit the notation and terminology used by McInnes and Pedersen (2013) as well as Zhu and Iglesias (2017) in their research of semantic similarity (and relatedness) computation between concepts in knowledge graphs. In the introduced knowledge-based approaches to semantic similarity we will study a knowledge graph (KG), which is a directed labeled graph $G = (V, E, \tau)$, where V is a set of *nodes*, E is a set of *edges* between nodes and τ is a function $V \times V \rightarrow E$ that defines all spo-triples in G. An edge e connecting two consecutive nodes $v_k, v_{k+1} \in V$ will be denoted as $e = < v_k, v_{k+1} > \in E$. Nodes of the knowledge graph contain concepts (like *Animal* or *Car* in Figure 5) as well as their instances (like *Missy* in Figure 5). Edges describe the relations (like *isA* and *has* in Figure 5) between concepts and instances. The semantic similarity between two concepts $c_i, c_j \in V$ will be denoted as $sim(c_i, c_j)$.

Path method

A path connecting c_i and c_j will be denoted as $P(c_i, c_j) = \{ c_i, < c_i, v_k >, v_k , < v_k, v_{k+1} >, v_{k+1}, \dots , c_j \}$. Two concepts can be connected via various paths of various lengths, the shortest path length denoted as $minpath(c_i, c_j)$.

[7] RDF: https://www.w3.org/RDF
[8] RDF Schema: https://www.w3.org/TR/rdf-schema
[9] OWL: https://www.w3.org/TR/owl-ref

The *path method*, originally called *distance method* by its developers Rada et al. (1989), simply uses the shortest path length to present the semantic distance between concepts. Semantic similarity is inversely proportional to the path length and indicated as

$$sim_{path}(c_i, c_j) = \frac{1}{1 + minpath(c_i, c_j)}.$$

The notation of *path* in the subscript refers to path method. The shorter the path, the more semantically similar the concepts are. The addition of one in the nominator prevents dividing by zero if $c_i = c_j$. As a matter of fact, path method is all about counting the edges in the shortest path between the concepts.

Wup method

Wu and Palmer (1994) (therefore the abbreviation *wup*) use the Least Common Subsumer (LCS) in their approach to semantic similarity computation. LCS is the nearest shared ancestor of the two concepts, a common hypernym. In Figure 5, the LCS of *dog* and *cat* is *mammal*, and the LCS of *dog* and *ant* is *animal*. We will denote LCS between concepts c_i and c_j as $lcs(c_i,c_j)$. Wup uses the following formula to measure the semantic similarity between two concepts

$$sim_{wup}(c_i, c_j) = \frac{2\ depth(lcs(c_i, c_j))}{depth(c_i)\ +\ depth(c_j)},$$

where $depth(c_i)$ is the number of edges from root to node c_i. What this adds to the path method is that also the specificity of the concepts affects the similarity measure, not just the path length between them. If we look at Figure 5, the path length between *Object* and *Animal* is equal to the path length between *Cat* and *Dog*, but the latter pair gets higher similarity points for being more specific, i.e., locating deeper in the taxonomy.

Li method

The approach by Li et al. (2003) combines the shortest path length and the depth of LCS in another way. Euler's number *e* along with parameters α and β, contribute to the shortest path length and depth of LCS respectively

$$sim_{li}(c_i, c_j) = e^{-\alpha\ minpath(c_i,c_j)} \frac{e^{\beta\ depth(lcs(ci,cj))}\ -\ e^{-\beta\ depth(lcs(ci,cj))}}{e^{\beta\ depth(lcs(ci,cj))}\ +\ e^{-\beta\ depth(lcs(ci,cj))}}.$$

The empirical optimal values for the parameters have been identified by the authors as α = 0,2 and β = 0,6.

Lch method

Shortest path is exploited in a method by Leacock and Chodorow (1998) as well, but this time with a non-linear function. The similarity between two nodes is computed as follows

$$sim_{lch}(c_i, c_j) = -\log \frac{minpath(c_i, c_j)}{2\,D},$$

where *D* is the maximum depth of the whole concept taxonomy tree. In other words, *D* is the number of edges from the root node to the furthest leave node. So, lch similarity measure is relative to the taxonomy height.

3.4.2 Information content based methods

The methods introduced above were all path-based (or edge-based) methods, relying only on path lengths on taxonomy tree traversals. There are also semantic similarity metrics that exploit the *information content* (IC) given by the word probabilities (frequencies) in corpus. While exploiting corpora, IC-based (or node-based) methods are knowledge-based methods and should not be confused with actual corpus-based methods discussed in Chapter 3.5.

Res method

The basic argumentation of information theory originally proposed by Shannon (1948) says that information content (IC) of a concept *c* can be quantified as a negative log-likelihood of its probability

$$IC(c) = -\log p(c).$$

If we have a corpus of *N* concepts, the probability to encounter concept *c* in the corpus is simply

$$p(c) = \frac{freq(c)}{N}.$$

When the probability of encountering the concept in corpus increases, its IC decreases. Resnik (1995) modified IC to be used as a similarity measure in the following manner

$$sim_{res}(c_i, c_j) = IC(lcs(c_i, c_j)) = -\log p(lcs(c_i, c_j)).$$

The choice of a logarithmic base corresponds to the unit for measuring information. Originally Shannon (1948) used base 2, in which case the resulting similarity is given in *bits* (or *shannons*). If natural logarithm (base *e*) is used instead, the result will be in units called *nats*.

According to the above formula, the similarity between two concepts equals the information content of their LCS, the nearest common ancestor (hypernym) of the concepts. It is important to remember that taxonomy wise the ancestor includes all its descendants, hence

$$p(c) = p(c') + \sum_{d \in descendant(c')} p(d),$$

where *c'* denotes the concept (word) *c* per se without its descendants. The semantic similarity between *dog* and *ant* measured by the res method equals the probability to encounter any concept from any of the subclasses of *animal* in the corpus (*animal* being the nearest common hypernym of *dog* and *ant*). If the concepts do not share any other common hypernym than the root of the taxonomy, their similarity is 0, since $p(root) = 1$ and $\log(1) = 0$.

Jcn method

Jiang and Conrath (1997) extended res method by including the IC of the individual concepts. They measured the distance between concepts as

$$dist(c_i, c_j) = IC(c_i) + IC(c_j) - 2\,IC(lcs(c_i, c_j)),$$

which gives similarity as its inverse

$$sim_{jcn}(c_i, c_j) = \frac{1}{IC(c_i) + IC(c_j) - 2\,IC(lcs(c_i, c_j))}.$$

Lin method

Lin (1998) used the same pieces as Jiang & Conrath (1997), when introducing his solution

$$sim_{lin}(c_i, c_j) = \frac{2\,IC(lcs(c_i, c_j))}{IC(c_i) + IC(c_j)}.$$

If we compare this to the wup method introduced earlier, we will find them to be similar except for the use of IC rather than the depth of the concepts.

The discussed methods have captured the taxonomical semantic similarity between concepts in ontologies. This can be extended to instances as well, since concepts can be viewed as semantic classes for their instances (Zhu and Iglesias, 2017). Thus, in Figure 5 the semantic similarity between instances *Missy* and *Lassie* can be measured by calculating the semantic similarity between their respective concepts, *Cat* and *Dog*.

3.4.3 Semantic relatedness methods

In addition to semantic similarity, also semantic relatedness can be extracted from knowledge graphs. Relatedness does not rely only on hierarchical (*isA*) relationships between concepts but also considers, for example, the properties (*has*) of classes or relations between instances (*isSiblingTo*). In other words, the importance of an edge between two nodes also has a measure. Recent knowledge-based methods have been proposing metrics

for measuring semantic relatedness. A couple of them will be introduced shortly without going into further details of the relatedness computation.

Hulpus et al. (2015) introduced a path-based semantic relatedness method, that uses social network analysis technique to measure the effectiveness of a path connecting instances. This is used together with the exclusivity metric that specifies the relative importance of the relations connecting these instances. The method follows two principles:

1. the shorter the path between instances, the higher the relatedness between them,
2. the relatively more important the relations between instances, the higher the relatedness between them.

In their approach, Han et al. (2012) introduced the Concept Level Association knowledge (CAK) to represent the knowledge essential for human language understanding. Such knowledge includes facts like *the birds can fly but trees cannot*, and *database table is not a kitchen table*. Automatic CAK learning will be obtained by computed statistical associations between instances, which base on the occurrences and co-occurrences of nodes and edges (terms and relations) in ontologies.

Schuhmacher and Ponzetto (2014) used Combined Information Content (CombIC) to derive weights for edges representing properties in the knowledge graph. The weights should capture the degree of associativity between concepts, as well as their different levels of specificity.

Several other knowledge-based approaches to measuring semantic similarity or relatedness between words have been introduced in the literature. The interested reader is suggested to explore Harispe et al. (2017), who give a quite exhaustive, chronological listing of them.

During the chapter, we have mostly discussed similarity between *concepts*, although the title says, "similarity between *words*". It is true, that words can have several meanings and therefore appear several times in the taxonomy as different concepts. So, how do we compare the similarity considering these polysemic words? What sense of the word do we choose? Tversky (1977) demonstrated in his psychological studies, that humans pay more attention to word similarities than their differences while rating the similarity between two words. Hence, *point* and *comma* are rated similar, despite *point* has many other kinds of meanings. The same principal is exploited in computational word similarity, i.e., taking the maximal similarity score over all concepts that are senses of the word

$$sim_{word}(w_i, w_j) = \max_{c_i \in s(w_i),\, c_j \in s(w_j)} sim_{concept}(c_i, c_j),$$

where $s(w)$ denotes a set of concepts that are senses of the word w (Sánchez *et al.*, 2012; Zhu and Iglesias, 2017).

The presented knowledge-based semantic similarity metrics have been reported to perform well in WordNet[10]. WordNet is integrated in popular knowledge graphs such as DBpedia[11], YAGO[12] and BablNet[13] enabling efficient semantic similarity computation (Zhu and Iglesias, 2017). WordNet is a lexical database of English language, where words are grouped into sets of cognitive synonyms called synsets, each expressing a distinct concept. These concepts are linked by semantic (and lexical) hierarchical relations (Princeton University, 2010).

In general, knowledge-based methods are computationally effective. They do not require complex and time-consuming semantic language model building, like corpus-based approaches do. The language model is already built into the ontology. Although, not every domain has its own ontology describing the elements to compare. That is a strong limitation. However, there is plenty of literature available for knowledge base generation and nothing prevents us from creating our own ontology to serve our purpose. (Harispe *et al.*, 2017)

3.5 Corpus-based semantic similarity methods

The lack of suitable ontology might drive us to find other solutions to handle semantic likeness between words. *Corpus-based methods* do not need anything but a vast amount of text for the task. The legendary quote from an English linguist John R. Firth (1957) captures the essence: "*You shall know a word by the company it keeps*". Words in similar *contexts* tend to indicate similar meanings. This idea derived originally from the research of distributional structure of language by Harris (1954), and has later become known as the distributional hypothesis (Sahlgren, 2008). Building on that we can utilize the similarity in word distributions to estimate the similarity in word meanings. Corpus-based semantic similarity approaches rely on this intuition.

Corpus-based methods identify the degree of likeness between words using information entirely derived from large corpora, i.e., large collections of text data (Mihalcea, Corley and Strapparava, 2006). There are several free public corpora to be taken advantage of in corpus-based NLP tasks. At the time of writing the thesis, corpus lists can be found, for example, at Kielipankki[14] website (Finnish corpora) and Nicolas Iderhoff Github[15] website (English corpora).

[10] WordNet, a large lexical database of English: https://wordnet.princeton.edu
[11] DBpedia, towards a public data infrastructure for a large, multilingual, semantic knowledge graph: https://wiki.dbpedia.org
[12] YAGO, a large semantic knowledge base, derived from Wikipedia, WordNet, WikiData, GeoNames, and other data sources: https://github.com/yago-naga/yago3
[13] BabelNet, a multilingual encyclopedic dictionary and a semantic network: https://babelnet.org
[14] Kielipankki, The Language Bank of Finland: https://www.kielipankki.fi/corpora
[15] Nicholas Niderhoff Github site: https://github.com/niderhoff/nlp-datasets

Since corpus-based approaches can rely on various vast corpora, they usually have better coverage of vocabulary than knowledge-based approaches, that can cover only the concepts included in the knowledge graph (Zhu and Iglesias, 2017). Corpus-based methods reflect all kinds of relations between words, both hierarchical and non-hierarchical. Hence, they principally measure semantic relatedness between words. Corpus-based methods do not take into account different meanings of polysemic words. For example, *point* is just a word, and does not represent any concept (such as punctuation, spot or fact) more than another. However, it could be represented in a way, that reflects its various meanings, as we will find out.

3.5.1 Distributed word representation

In the applications, the words are usually represented using well-known mathematical objects such as sets, vectors, probability distributions or nodes in the graphs. The representation of the words naturally determines the way we compute the similarity between them. Knowledge-based methods rely on words as nodes in the graphs, as we saw in Chapter 3.4. Corpus-based methods, since relying on statistical computations, have to construct numerical vectors from words.

There are principally two ways to present words as vectors, a symbolic (also referred to as local) and distributed representation. For many years presenting words as atomic units, like symbols, was the predominant one. In the *symbolic representation*, the *k*:th word in the vocabulary is presented as a *one-hot* vector $\boldsymbol{x}$ as follows

$$\begin{cases} x_i = 1, & when\ i\ =\ k \\ x_i = 0, & otherwise, \end{cases}$$

where k stands for the ordinal of the word in the vocabulary and x_i is the i:th element of the vector $\boldsymbol{x}$. In other words, the k:th element in $\boldsymbol{x}$ is 1 and all other elements are 0. We can illustrate this with a simple example. Let us take the vocabulary of four words {*cat, dog, car, bike*}. The symbolic one-hot vector representation of the 3rd word in the vocabulary, *car*, is $[0\ 0\ 1\ 0]^T$. The dimension of the vector in this case is 4. More generally, in the vocabulary of size V, the symbolic representations of words are one-hot vectors of the length V.

Presenting words in a vocabulary as one-hot vectors has its virtues. It is very easy to implement and easy to understand. Unfortunately, it comes with a crucial failure considering our purpose. There is no decent measure for semantic similarity between words in the symbolic representation. A word *cat* is as close to a word *dog* as it is to a word *car*. They are just vectors with one entry as 1 and others as 0. If measured as Euclidian distance between the words, the distance of a given word from any another word will be $\sqrt{1+1} = \sqrt{2}$ (Goodfellow, Bengio and Courville, 2017).

The other way to present words is as *continuous distributed vectors*, which Bengio (2003) aptly calls word *feature* vectors. Let us review the previous four-word vocabulary {*cat, dog, car, bike*}. Figure 6 presents the words of the vocabulary as one hot vectors and distributed vectors in 4-dimensional vector space. The difference between these presentations is that in the symbolic presentation the dimensions of the vector are always the objects (the words) themselves, while in the distributed representation the dimensions are features of the object. In this case we have selected the following features:

1. is the object a *pet* (yes = 1, no = 0),
2. is the object a *vehicle* (yes = 1, no = 0),
3. how many *legs* does the object have,
4. how many *wheels* does the object have.

Of course, we could have selected any number of dimensions with any features we desired.

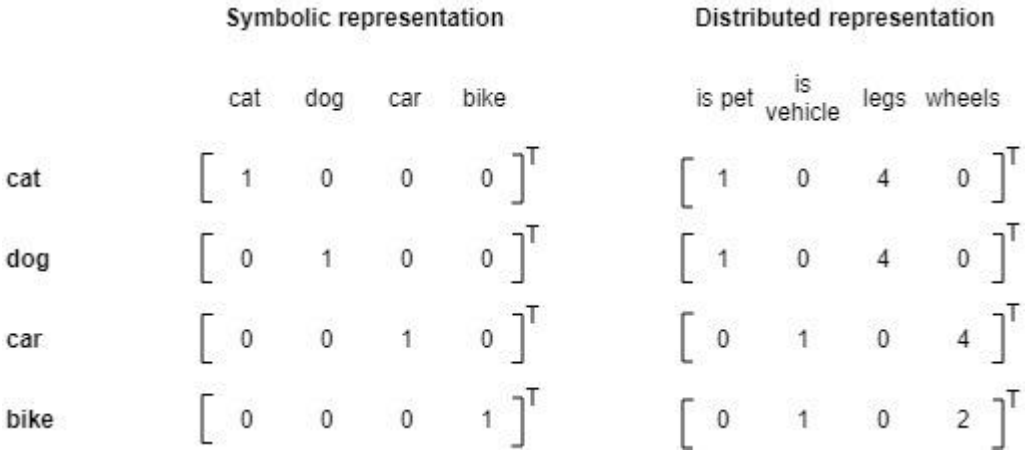

Figure 6. *Words of a four-word vocabulary presented as symbolic one-hot vectors and distributed vectors.*

We can notice how our distributed representation captures the semantic similarity between cat and dog ($[1\ 0\ 4\ 0]^T$ and $[1\ 0\ 4\ 0]^T$), as it does with car and bike ($[0\ 1\ 0\ 4]^T$ and $[0\ 1\ 0\ 2]^T$. Both vector pairs are close to each other in the vector space. In fact, cat and dog share the same distributed vector representation.

In our example, we chose the four features for our distributed representation. We also gave them names to help visualize the idea: *pet, vehicle, legs* and *wheels*. By these features we found the similarity between words. Cats and dogs were pets with legs, cars and bikes were vehicles with wheels. Of course, in real life's NLP tasks the vocabularies are much larger and we need higher dimensionality for our distributed representation to capture the relevant features of the word, typically 300 dimensions (Church, 2017). Sometimes these features are understandable for a human but usually they are not (Landauer, Folt and Laham, 1998). Neither are they discrete as in our example, but continuous-valued

(Bengio, 2008). So, the concept could be somewhat pet-like although it is not a pet, for example, "pet-likeness" could be 0,4 for *squirrel* and 0,9 for *cat*.

These features may seem rather mystical at first. However, they can be learned by the help of statistics related techniques. According to Landauer et al. (1998), the features can be considered as *latent* features. We can imagine that each dimension in feature space corresponds to a semantic or grammatical characteristic of words (Bengio, 2008). Let us get back to our previously mentioned example of the polysemic word *point*. Its distributed vector representation could learn various meanings of *point* as features. For example, one feature could be its "fact-likeness", one could be "spot-likeness" and one could be "punctuation-likeness".

3.5.2 LSA, Latent Semantic Analysis

Latent Semantic Analysis (LSA) was introduced as a new approach to automatic information retrieval and indexing by Deerwester et al. (1990). However, Landauer et al. (1998) were the first to discuss LSA in the context of word meanings and word similarity measuring. As the name implies, LSA tries to discover the hidden semantic features of the words. In order to succeed, LSA uses dimensionality reduction to extract semantics from term occurrences in a corpus. This is executed by Singular Value Decomposition (SVD) on the *term-by-document* matrix $\boldsymbol{T}$ representing the corpus. In literature, term-by-document matrix is also referred to as word-by-document, word-by-context, document-word or term-context matrix. Each row in $\boldsymbol{T}$ stands for a unique term and each column stands for a document (see Table 1). Documents are represented as bags of words (BOWs), where only the word counts are relevant, not the order of the words. This representation defines the context of a word as a document in which the word occurs. It was originally introduced as Vector Space Model by Salton et al. (1975).

We can demonstrate creating term-by-document matrix $\boldsymbol{T}$ with a simple example of the following four short documents:

1. D1: *Cats* and *ants* have *legs*.
2. D2: *Dogs* and *cats* have *fur*.
3. D3: *Cars* and *lorries* have a *wheel*.
4. D4: *Cars* and *bikes* have *tires*.

After preprocessing, for example, removing stop words and normalizing the words, the vocabulary will contain ten terms: {*cat*, *ant*, *leg*, *dog*, *fur*, *car*, *lorry*, *wheel*, *bike*, *tire*}. Term-by-document matrix constructed from documents D1, …, D4 appears in Table 1 and tells us how many times each word appears in each document. This time, the maximum count of appearances in documents happens to be 1, but naturally in larger corpora it is much higher.

Table 1. *A term-by-document matrix of four documents and a ten-word vocabulary.*

	D1	**D2**	**D3**	**D4**
cat	1	1	0	0
ant	1	0	0	0
leg	1	0	0	0
dog	0	1	0	0
fur	0	1	0	0
car	0	0	1	1
lorry	0	0	1	0
wheel	0	0	1	0
bike	0	0	0	1
tire	0	0	0	1

We can also reflect term-by-document matrix more generally. If we broaden the concept of document to cover all kinds of contexts, we will get a general term-by-context matrix. The context could be, for example, a window of ten consecutive words. Now each word vector would be defined by term occurrences inside this context window.

SVD is a matrix operation, that can be applied to any rectangular $m \times n$ matrix to find correlations among its rows and columns. SVD decomposes the term-by-document matrix $\boldsymbol{T}$ into three matrices

$$\boldsymbol{T} = \boldsymbol{U}\,\boldsymbol{\Sigma}_k\,\boldsymbol{V}^{\mathrm{T}},$$

where $\boldsymbol{\Sigma}_k$ is the rectangular diagonal matrix containing $k = \min(m, n)$ *singular values* of $\boldsymbol{T}$, $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_k$, and $\boldsymbol{U}$ and $\boldsymbol{V}$ are column-orthogonal matrices. The sizes of $\boldsymbol{U}$, $\boldsymbol{\Sigma}_k$ and $\boldsymbol{V}^{\mathrm{T}}$ are respectively $m \times m$, $m \times n$ and $n \times n$ in full SVD and $m \times n$, $n \times n$ and $n \times n$ in reduced SVD (or thin SVD) assuming $m > n$. The condition $m > n$ applies if the vocabulary size is greater than the number of documents considered. In such cases LSA uses the reduced form (Landauer, Folt and Laham, 1998). The reduced SVD decomposed from the data in Table 1 is

$$\boldsymbol{T} = \boldsymbol{U} \quad \boldsymbol{\Sigma}_k \quad \boldsymbol{V}^{\mathrm{T}} =$$

$$\begin{pmatrix} 0{,}71 & 0 & 0 & 0 \\ 0{,}35 & 0 & 0{,}50 & 0 \\ 0{,}35 & 0 & 0{,}50 & 0 \\ 0{,}35 & 0 & -0{,}50 & 0 \\ 0{,}35 & 0 & -0{,}50 & 0 \\ 0 & 0{,}71 & 0 & 0 \\ 0 & 0{,}35 & 0 & -0{,}5 \\ 0 & 0{,}35 & 0 & -0{,}5 \\ 0 & 0{,}35 & 0 & 0{,}5 \\ 0 & 0{,}35 & 0 & 0{,}5 \end{pmatrix} \quad \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1{,}41 & 0 \\ 0 & 0 & 0 & 1{,}41 \end{pmatrix} \quad \begin{pmatrix} 0{,}71 & 0{,}71 & 0 & 0 \\ 0 & 0 & 0{,}71 & 0{,}71 \\ 0{,}71 & -0{,}71 & 0 & 0 \\ 0 & 0 & -0{,}71 & 0{,}71 \end{pmatrix}.$$

If we take only the *k'* largest singular values and replace the other singular values by zero, we will obtain a least-squares best fit approximation $\boldsymbol{T}'$ of the original $\boldsymbol{T}$

$$\boldsymbol{T}' = \boldsymbol{U}\, \boldsymbol{\Sigma}_{k'}\, \boldsymbol{V}^{\mathrm{T}}.$$

SVD identifies (and orders) the dimensions along which datapoints show the most variation (Harispe *et al.*, 2017). The two largest singular values of the term-by-document matrix $\boldsymbol{T}$ in Table 1 are both 2 (indicated on gray background in $\boldsymbol{\Sigma}_k$ of the SVD above). In Table 2 we will see the approximation of the original $\boldsymbol{T}$ obtained by using only these two largest singular values (and replacing other singular values by zero).

Table 2. *The approximation of the original term-by-document matrix obtained by its two largest singular values.*

	D1	**D2**	**D3**	**D4**
cat	1	1	0	0
ant	0,5	0,5	0	0
leg	0,5	0,5	0	0
dog	0,5	0,5	0	0
fur	0,5	0,5	0	0
car	0	0	1	1
lorry	0	0	0,5	0,5
wheel	0	0	0,5	0,5
bike	0	0	0,5	0,5
tire	0	0	0,5	0,5

How does this representation reveal any more similarity between the words than the original? Landauer et al. (1998) interpret this in a human way. By taking the two largest singular values, it is decided that every word consists of two abstract features (some amount

of feature 1 and some amount of feature 2). If we consider each document as a context of the word, the representation tells the *probability of the word appearing in that context*. The probability of *dog* appearing in context D1 is 0,5, even though it did not appear in document D1 originally. So, dog might have legs.

We can also think of the rows in the term-by-document matrix as vector presentations of the terms. Let us compare *ant* and *dog* to each other. They are both animals that intuitively could appear in similar contexts. However, their correlation (Pearson correlation co-efficient) in the original representation is – 0,33, and in the approximation, it is 1. In other words, they are equivalent vectors in this simple presentation. Hence, the reduced dimensional representation succeeds in capturing the semantic relatedness between the words.

Of course, this example of only four documents and ten words is an oversimplified case in many ways, but it gives us a hint of how LSA performs. In real life cases all words are not equally important. The importance of the word must also be taken into account. As stated before, when the probability of encountering the concept in corpus increases, its IC (information content) decreases and vice versa. Since *ant* is less probable word than *cat* in our corpus, we should give its appearance a greater importance.

To put all the words on the same line before performing SVD, their overall frequency in the corpus must be computed as well, not only their frequency in a given context. We should construct a weighted term-by-document matrix instead of the matrix including only the raw word counts. When introducing LSA, Landauer et al. (1998) used the information theoretic measure called *entropy* (see Shannon 1948) for weighting. Entropy is closely related to IC. The word frequencies were first converted to their logarithms and then divided by the word entropy. The entropy H for the word w_i is

$$H(w_i) = -p(w_i)\log p(w_i) = -p(w_i)IC(w_i).$$

Hence, the weighted term-by-document matrix entry T_{ij} (entry for the word w_i in document j) can be computed as follows

$$T_{ij} = -\frac{\log freq(w_{ij})}{p(w_i)\log p(w_i)},$$

where $freq(w_{ij})$ is the count of word w_i in document j. This weighs the occurrence of each word by an estimate of its importance. There are several other ways to execute the weighting, perhaps the information retrieval related *TF-IDF* (Term Frequency – Inverse Document Frequency, N.B. dash and hyphen do not refer to subtraction in this case) method introduced by Salton and McGill (1983) as the most popular one alongside the entropy method. In the basic approach TF (term frequency) for the word w_i in the document j is simply its word count, denoted here TF_{ij}. The number of documents that contain the word (document frequency) is denoted DF_i and it defines the inverse document frequency (IDF) as follows

$$IDF_i = \log \frac{N}{DF_i},$$

where N is the total number of documents. The TF-IDF weighting is conducted as a product of TF and IDF

$$TF - IDF_{ij} = TF_{ij}\, IDF_i.$$

Hence, also this method gives a high weight if the word is frequent in a particular document, but rare otherwise. Also more elegant versions of TF-IDF exist (Manning, Raghavan and Schutze, 2009). Nakov et al. (2001) have explored the impact of several weighting methods on LSA performance.

When the weighted entries for word vectors have been computed, the similarity between them must be evaluated. Landauer et al. (1998) used cosine distance for the word similarity computing. *Cosine similarity* of word vectors $\boldsymbol{w_i}$ and $\boldsymbol{w_j}$ is the cosine of the angle between them

$$sim_{cos}(\boldsymbol{w_i}, \boldsymbol{w_j}) = \cos(\boldsymbol{w_i}, \boldsymbol{w_j}) = \frac{\boldsymbol{w_i} \cdot \boldsymbol{w_j}}{\|\boldsymbol{w_i}\| \|\boldsymbol{w_j}\|},$$

with the dot product of the vectors $\boldsymbol{w_i}$ and $\boldsymbol{w_j}$ in the numerator and the product of their Euclidean norms in the nominator.

LSA is a classical method and acts as a good introduction to modern approaches. It includes many key features of the recent corpus-based similarity methods:

- LSA uses distributed representation for words and cosine similarity for computing the likeness (similarity or relatedness) between words.
- LSA learns meanings of words from the same data as humans learn: a large corpus.
- LSA is able to find latent features that construct the likeness between words mimicking human cognition.

It is interesting that while summarizing their introduction to LSA, Landauer at al. (1998) commented: "*It is hard to imagine that LSA could have simulated the impressive range of meaning-based human cognitive phenomena that it has unless it is doing something analogous to what humans do.*" This seems to be quite what the modern approaches try to tackle by the help of neural networks.

3.5.3 Neural network models

Language models use a slightly different approach from LSA to word similarity computing. A language model defines a probability distribution over a word sequence in a natural language. This is typically performed by predicting the next word given the preceding

ones. Many NLP applications, such as speech recognition and machine translation, use this kind of technique. The evolution of language models has led to state-of-the-art continuous distributed vector representations for words. We should begin from the initial language model, the *n-gram* model, to understand why and how this happened.

History of language models

N-gram based language models have been the dominant statistical language models until the turn of the millennium (Bengio, 2008). In this context, n-grams are sequences of *n* words (distinguished from character n-grams, which will be considered later). Shortest n-grams are usually referred as unigrams (1-grams), bigrams (2-grams) and trigrams (3-grams). N-gram models use *n*–1 order Markov model in predicting the probability of the *t*:th word in a sequence. In other words, only the *n*–1 preceding words matter and the words prior to that can be forgotten totally (Manning and Schütze, 1999).

A simple example clarifies the case. We want to estimate the probability of the word *meows* following the sequence "*my cat*", denoted *p*(*meows* | "*my cat*"). First, we must count how many times the sequence "*my cat*" occurs in the corpus. Second, we calculate how many times the word "*meows*" follows the sequence "*my cat*" in the corpus, i.e., the occurrences of the sequence "*my cat meows*" (3-gram). From the ratio of these two we will get the desired probability

$$p(meows \mid "my\ cat") = p("my\ cat\ meows") \ = \frac{freq("my\ cat\ meows")}{freq("my\ cat")}.$$

This is the simple basis of statistical language models. The previous formula using 3-gram model can be generalized to all n-gram models as follows:

$$p(w_t \mid w_{t-(n-1)}, w_{t-(n-2)}, \ldots, w_{t-1}) \ = \frac{freq(w_{t-(n-1)}, w_{t-(n-2)}, \ldots, w_{t-1}, w_t)}{freq(w_{t-(n-1)}, w_{t-(n-2)}, \ldots, w_{t-1})}.$$

When we define the probability distribution over a longer sequence of words, we can exploit the chain rule of probability (a consequence of Bayes theorem). The probability of a sentence is the product of conditional probabilities of each word given the preceding ones

$$p(w_1, \ldots, w_t) \ = \ p(w_1)\ p(w_2|w_1)\ p(w_3|w_1, w_2) \ldots\ p(w_t|w_1, \ldots, w_{t-1}),$$

where w_i denotes the *i*:th word of the sentence (Bengio, 2008).

Because models based on n-grams use only *n* – 1 word context for each conditional probability (*n*:th word given the *n*–1 preceding ones), the above formula applied to, for example, trigrams (*n* = 3, context = 2 words) is

$$p(w_1, \ldots, w_t) = p(w_1)\ p(w_2|w_1)\ p(w_3|w_1, w_2)\, p(w_4|w_2, w_3) \ldots\ p(w_t|w_{t-2}, w_{t-1})$$
$$= p(w_1)\ p(w_2|w_1) \prod_{k=3}^{t} p(w_k|w_{k-2}, w_{k-1}).$$

However, w_1 and w_2 do not actually have this 2-word context and are usually discarded (Goodfellow, Bengio and Courville, 2017). This leads to the general formula

$$p(w_1, \ldots, w_t) = \prod_{k=n}^{t} p(w_k|w_{k-(n-1)}, \ldots, w_{k-1}).$$

Bengio (2008) summarizes, that n-gram techniques are non-parametric machine learning algorithms for the next word prediction which are based on storing and combining frequency counts of word subsequences of *n* words and shorter. While we have these counts stored, we can estimate the probability of any sentence in the corpus using the previous formulas.

However, we often need to estimate probabilities of sequences never met in corpus (which is our training set for the machine learning model). New sequences are also possible, although the above introduced formula does not generalize on them at all. It gives zero (numerator = 0) or undefined (denominator = 0) probabilities. Several methods were developed to fix this (Katz, 1987; Goodfellow, Bengio and Courville, 2017). For example, back-off methods looked up the lower order n-grams (that is why we need to store also sequences shorter than *n*–1 words) and estimated the probabilities according to them, and smoothing methods added non-zero probability mass to all possible next word probability values.

The improvements did not fix the whole generalization problem. The n-gram model still did not capture the similarity of words and hence did not generalize well on semantically similar sentences. N-gram model is a local predictor looking for the nearest local neighbor word. However, the words are represented as one-hot vectors, and they all have exactly the same distance from each other (see Chapter 3.5.1). N-gram model did not see the similarity between sentences "*I love cats*" and "*I love kittens*". Class-based language models were supposed to overcome this problem. They introduced the notion of word categories for statistically similar words based on word co-occurrence frequencies with other words. Models then used the word class instead of the word per se, while predicting the next word probabilities of sequences. A lot of information was lost, as all words did not have distinctive representations. *Kitten* is not the same word as *cat*, despite their semantic similarity. (Goodfellow, Bengio and Courville, 2017)

Problems with similar words are not the only ones, though. Perhaps the most restrictive fault with n-gram models is the *curse of dimensionality*. In statistical language models we deal with word distributions. Consider we have a natural language vocabulary of 100 000

words and we want to model the joint distribution of 10 consecutive words. Bengio et al. (2003) point out, that our n-gram language model with symbolic (one-hot) word representations needs to relate each training sentence to every possible similar sentence. There will be potentially $100\,000^{10} - 1 = 10^{50} - 1$ free parameters. And in NLP tasks vocabulary of 100 000 words is a small one. What if we have a one-billion-word vocabulary or even larger? That is the curse of dimensionality. It is impossible to have enough training examples for every possible sentence.

To overcome these two problems, i.e., the curse of dimensionality and failure to capture the semantic similarity between words, we need a language model, that is somehow able to share knowledge between words. Distributed word representations respond exactly to this need. Bengio (2003) introduced a neural probabilistic language model, which learned continuous distributed vector representations for words and hence allowed each training sentence to inform the model about an exponential number of semantically similar sentences.

Mikolov (2009), the father of the word2vec model (discussed in Chapter 3.5.4), considers Bengio as an originator of the modern neural network language model. However, distributed representation of words as continuous vectors learned by a neural network was not a new invention at the time. In the late 80's Geoffrey Hinton with his colleagues introduced how distributed representations can be learned by back propagating through neural network (Hinton, McClelland and Rumelhart, 1986; Rumelhart, Hinton and Williams, 1986). Neural networks need a lot of computing, and lack of computing power those days prevented the model from gaining more popularity. So, Bengio (2003) was the one who succeeded in demonstrating how his neural network model surpassed standard n-gram models on statistical language modeling tasks.

General idea behind neural network language models

When we deal with extremely complex phenomena, like natural language processing and image recognition, *neural networks* and *deep learning* are the state-of-the-art techniques to reach the best solution. They are wide and extremely versatile subjects to cover briefly. The interested reader is recommended to explore, for example, Goodfellow et al. (2017), who give a thorough introduction to methods and applications in the field. Language has been modeled with several kinds of neural networks. The most simple and effective implementations use a feed forward network that learns by a supervised learning method (Mikolov, Corrado, *et al.*, 2013; Church, 2017).

We will not go any deeper into the details of earlier neural network models here. The most important details will be covered largely enough in the next chapter, while we explore word2vec, probably the most popular neural network driven model at the time (Church, 2017). However, in order to follow this, the reader must be briefly introduced to the idea of neural networks. It should be emphasized, that what follows is a really simplified summary of their architecture and learning.

In Figure 7 we have a simple example of a feedforward neural network, also called *multilayer perceptron* (MLP). Network consists of *neurons* which form the network layers: input layer, hidden layer(s) and output layer (which could also consist of several neurons). In fully connected MLP the information moves from layer to layer and every neuron from the preceding layer is *connected* to every neuron in the next layer. These connections (synapses) have weights, denoted by W_{ij} and W'_i in the figure.

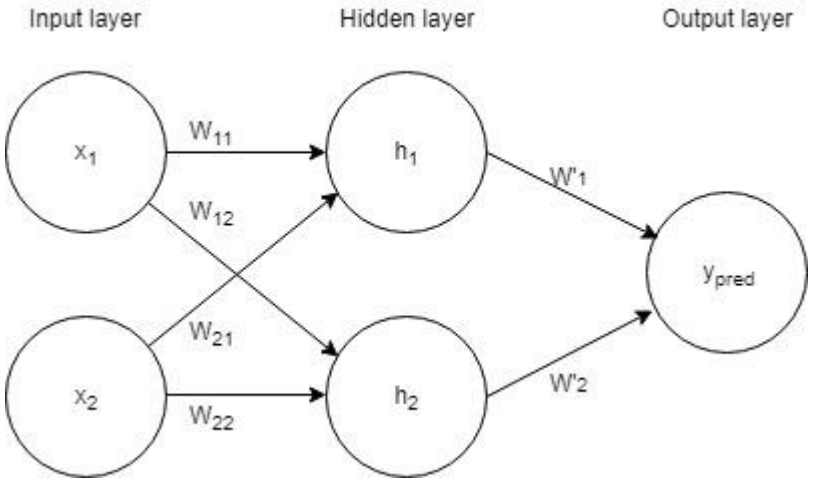

Figure 7. *A simple feedforward neural network, multilayer perceptron (MLP). Biases and activation functions are not indicated in the figure.*

Let us take a math test score prediction as an example. The input $\boldsymbol{x}$ in Figure 7 is a 2-element vector, and x_1 could denote hours spent doing homework and x_2 hours spent preparing for the test. Output y denotes the test result. The weights W_{ij} and W'_i could be considered as measures which tell us how much each connection influences the output. The weights W_{ij} map the input to the hidden layer and weights W'_i map the output of the hidden layer to the network output. In this simplified case our network is trying to predict the output y_{pred} from the input $\boldsymbol{x}$ via the hidden layer $\boldsymbol{h}$ as follows

$$\boldsymbol{h} = \boldsymbol{W}^{\boldsymbol{T}} \boldsymbol{x}$$

$$y_{pred} = \boldsymbol{W'}^{\boldsymbol{T}} \boldsymbol{h}$$

$$y_{pred} = \boldsymbol{W'}^{\boldsymbol{T}} \boldsymbol{W}^{\boldsymbol{T}} \boldsymbol{x}\,.$$

This is the *forward phase* of our network. It feeds the input forward and computes values for every neuron in the hidden layer(s) and in the output layer. However, this function is linear and the real-life phenomena we try to predict with neural networks usually are not. For example, the math test score does not have to be a linear combination of hours spent on homework and preparing for the test. Hence, to break the linearity and make the network able to learn more complex functions, neural networks use activation functions and biases before hidden and output layer values are computed.

Activation function is applied to the output of a linear transformation to yield a nonlinear transformation. For example, if we use an activation function g_1 in the hidden layer of MLP in Figure 8, the output of $W_{11}\,x_1 + W_{21}\,x_2$ will become $g_1(W_{11}\,x_1 + W_{21}\,x_2)$, hence becoming the value of neuron h_1. *Bias* b_i can be added to every layer in the network to adjust its output, e.g, $W_{11}\,x_1 + W_{21}\,x_2 + b_1$, thus yielding the final value $h_1 = g_1(W_{11}\,x_1 + W_{21}\,x_2 + b_1)$, where b_1 is the bias for the input layer. Network will learn optimal bias terms while training (Goodfellow, Bengio and Courville, 2017). The recommended activation function for modern neural networks is the *rectified linear activation* function $g(z) = \max\{0, z\}$, which replaces all negative outputs with 0 (Glorot, Bordes and Bengio, 2011).

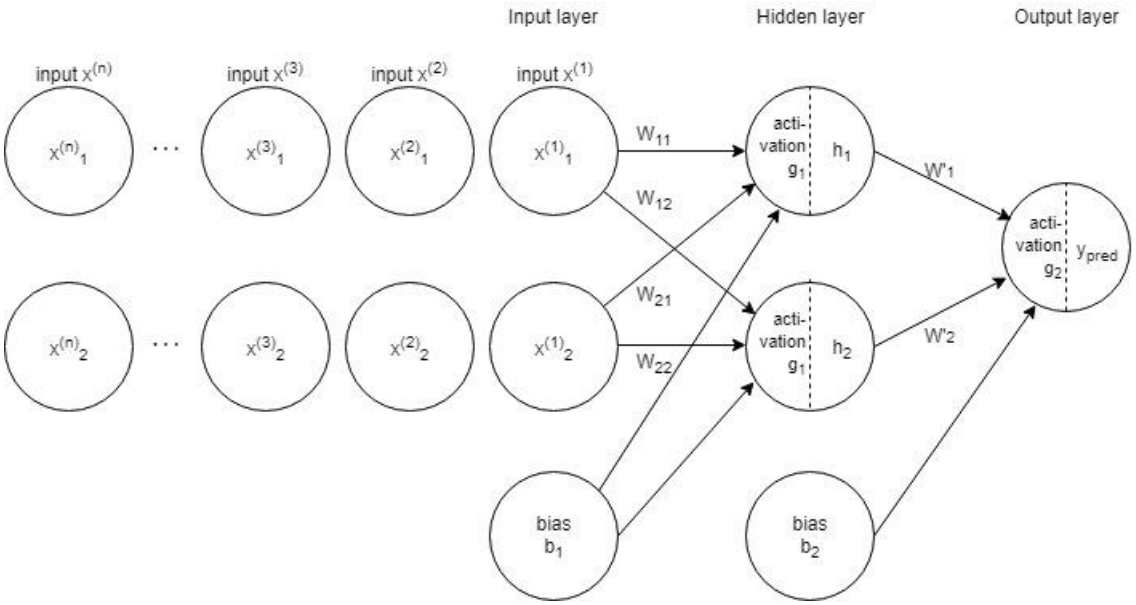

Figure 8. *A simple MLP with huge amount of training data (n vectors), biases b_i and activation functions g_i.*

Often the neural network is harnessed to perform a *classification* task. In other words, it classifies inputs into two or more output classes. In that case the output must be a vector of length equal to the number of the classes. The output needs to represent a probability distribution over these classes. Regarding binary classification, Goodfellow et al. (2017) recommend using *sigmoid* activation function before the output layer

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Sigmoid squashes the outputs near values 0 and 1. In the case of word classification, i.e., predicting the next word probabilities, we have a massive number of output classes. In this case, we can use a generalization of the sigmoid called *softmax* (Bengio *et al.*, 2003; Mikolov *et al.*, 2009). Softmax squashes the N-dimensional vector $\boldsymbol{z}$ of arbitrary real values to give a proper probability distribution summing up to 1

$$\sigma(\mathbf{z}_j) = \frac{e^{z_j}}{\sum_{n=1}^{N} e^{z_n}} \; for\; j \; = \; 1, \ldots, N.$$

Our network in Figure 7 and Figure 8 does not classify but computes us a predicted output, for example, a predicted math test score y_{pred} for each input vector. This probably differs a lot from the real output y_{real}, since we started the prediction with random weights. Of course, we want our prediction to be as close to real as possible. In other words, the prediction error $y_{real} - y_{pred}$ should be minimized. So, our weights $\boldsymbol{W}$ and $\boldsymbol{W'}$ need to be adjusted. It would not be reasonable just to guess some new weights until we reach an error close to zero. The optimal weights for the task should be *learned* by the network's learning algorithm while minimizing the error.

Our MLP in Figure 7 had only one input vector $\boldsymbol{x}$. To train the network make useful predictions we naturally need a massive amount of these vectors. This set is called our *training data*. In Figure 8 we see an illustration of all the input vectors $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \ldots, \boldsymbol{x}^{(n)}\}$ as they are fed into our network one after the other. The network first makes its predictions for our training data using its initial random parameters (weights and biases). In the end, it has gained some total error in its predictions, and this error should be minimized. To minimize the error, we must indicate it in the form of a differentiable function, since the most common learning algorithms for feedforward networks are gradient based. These functions are called *loss functions* or *cost functions*. For simplicity, we could use squared error $L = \sum \frac{1}{2}(y_{real} - y_{pred})^2$ as our loss function in this case. Various good loss functions exist, and the choice remains task specific. For example, in the case of multiclass classification of words, the negative log-likelihood is often used (Bengio *et al.*, 2003; Mikolov, Corrado, *et al.*, 2013). The minimum error will be in one of the critical points of the loss function, i.e., where its gradient is zero. The guiding idea of the learning algorithm is to gradually get closer to this minimum by adjusting the weights and biases of the network in small steps. Gradient of loss tells us whether we are moving towards the minimum or away from it. Naturally we hope to find the global minimum of the loss function, but this is usually difficult (see Figure 9). Goodfellow et al. (2017) introduce ways for learning algorithms to avoid poor local minima.

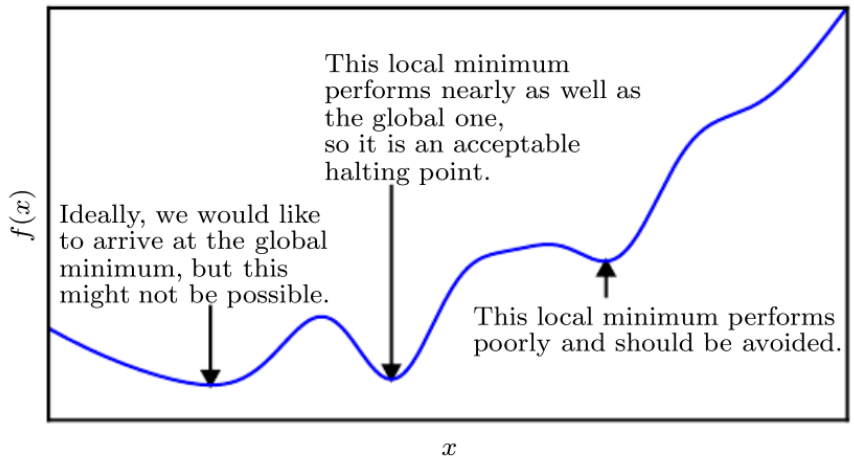

Figure 9. *Minima of loss function. (Goodfellow, Bengio and Courville, 2017, p. 85)*

The network knows how much each weight and bias should be adjusted by using a differentiation technique called *back propagation* (Rumelhart, Hinton and Williams, 1986). In back propagation the information from the loss L flows backwards through the network in order to compute the gradient needed for minimizing the error. This procedure is the *backward phase* of our network training. The forward phase calculated the values for every neuron in the network, and the backward phase calculates partial derivatives of the loss L with respect to every parameter (weight and bias) in the network step by step, starting from the last parameters.

Since back propagation is an essential part of the learning process, we will illustrate this with an example. In what follows we will consider updating the weight W'_1 in Figure 7 (N.B. not in Figure 8 for the sake of simplicity). To find out the effect of W'_1 in the total loss $L = \sum \frac{1}{2}(y_{real} - y_{pred})^2$ we can calculate its partial derivative exploiting the chain rule of calculus in the following way

$$\frac{\partial L}{\partial W'_1} = \frac{\partial L}{\partial y_{pred}} \frac{\partial y_{pred}}{\partial W'_1}.$$

If we calculate both factors on the right, we get

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial \frac{1}{2}(y_{real} - y_{pred})^2}{\partial y_{pred}} = (y_{real} - y_{pred})(-1) = y_{pred} - y_{real}$$

$$\frac{\partial y_{pred}}{\partial W'_1} = \frac{\partial W'_1 h_1}{\partial W'_1} = h_1$$

and the final partial derivative

$$\frac{\partial L}{\partial W'_1} = (y_{pred} - y_{real})\, h_1.$$

Our network computed and stored all the needed values in its forward phase, so this can be computed using them. If the calculated gradient is positive, we are going upwards away from the critical point, hence the weight should be decreased (and respectively increased if the gradient is negative). As mentioned before, we want to get closer to the minimum with small steps (to avoid missing it). Hence, we have a *learning rate* (extremely small factor) η, which controls the updates. The updated W'_1 will be

$$w_1 = W'_1 \pm \eta \frac{\partial L}{\partial w_1}.$$

Other weights are updated respectively. For example, to find out the impact of the weight W_{11} on the loss L we compute the partial derivative

$$\frac{\partial L}{\partial W_{11}} = \frac{\partial L}{\partial h_1}\frac{\partial h_1}{\partial W_{11}} = \frac{\partial L}{\partial W'_1}\frac{\partial W'_1}{\partial h_1}\frac{\partial h_1}{\partial W_{11}}$$

We have already calculated the first factor in the chain, i.e., the derivative with respect to the weight w_1 (the last parameter of our network). We can exploit these already computed derivatives while moving backward in our back propagation. Calculations become naturally much more complicated when our network has activation functions and biases and several output neurons, since we must update every parameter (every weight, every bias) in our network accordingly. The principle remains the same, though: we compute the partial derivative of the total loss with respect to the parameter and update the parameter so that we will gradually move closer to the minimum of the loss. We can control our progress by reducing our speed, which means gradually decreasing our learning rate η. Finally, we will reach the point our loss does not decrease significantly, and we can stop.

Using the above-mentioned techniques, feedforward neural network will become a *universal function approximator* (Goodfellow, Bengio and Courville, 2017). Given at least one hidden layer with enough neurons, it will be able to learn any function possible from its vast training data with its rectified linear functions. However, the neural network is not only supposed to memorize what it has seen, but to generalize to data totally unfamiliar to it. Hence, we must have a *test set* of examples our network has never seen. Usually, while training our network, the input set is divided into the training set (e.g., 80 %) and the test set (e.g., 20 %). This gives us the opportunity to test the generalization of our network to unknown examples with the test set. Naturally, our objective is to reduce the error for the test set. A trade-off between training error and test error will often remain: while reducing the test error, the training error will increase and vice versa. Goodfellow et al. (2017) propose several techniques, such as *weight penalty* or *dropout*, to accomplish good generalization of the network during the training.

In a nutshell, the network learns like this: first we train and optimize the network with our training set and back-propagation, then we test the network with our test set. This round is called *epoch*. We will repeat these epochs until our test error does not decrease significantly anymore. During the process, the network will learn proper parameters.

Neural network language models use above introduced feedforward neural networks (as well as other implementations, such as recurrent neural networks) for their next word predictions. The architecture for the model presented by Bengio et al. (2003) can be found in Figure 10. The general idea is to feed a sequence of previous n words into a network as an input, and make the network compute the next word probabilities as an output. The input words are fed into the network as one-hot vectors and projected to continuous distributed vectors by the mapping C. While the network learns the best possible next word predictions, the projection C simultaneously learns a feature vector (see Chapter 3.5.1) representation for every word. The network uses *tanh* (a rescaled sigmoid) activation function in the hidden layers to break the linearity and softmax in the output layer to give a proper probability distribution for the next word predictions. The network uses penalized log-likelihood as its loss function.

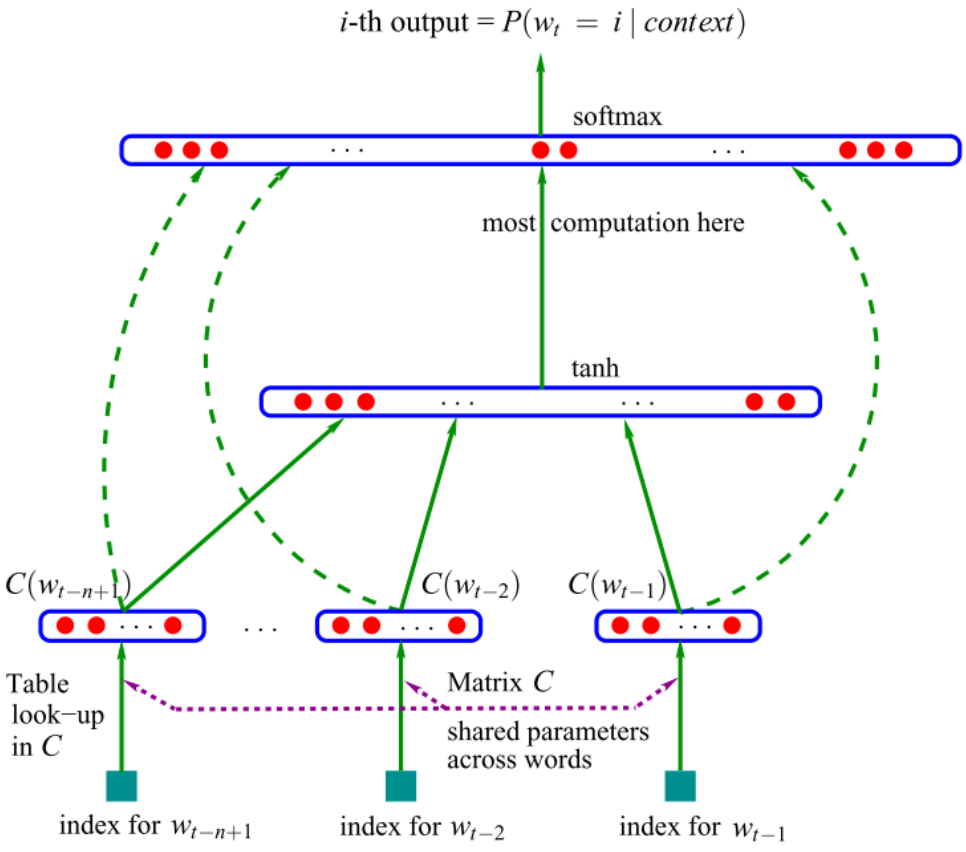

Figure 10. *Neural network architecture for the language model presented by Bengio et al. (2003, p. 1142)*

3.5.4 Word2vec word embeddings

The word feature vectors are often called *word embeddings*. Tomas Mikolov (2013) and his research team at Google introduced two efficient neural network architectures for learning word embeddings: *continuous bag-of-words* (CBOW) model and *continuous skip-gram* model. They are both often referred to as *word2vec model*.

Word2vec has achieved a very stable position in NLP tasks since. There are at least two reasons for its popularity: word2vec is rather simple and as an open source product accessible to anyone (Church, 2017). Hence, word2vec is implementable for everyone with adequate skills, and without skills one can utilize various easy-to-use implementations for different programming languages, such as Java/Scala, Python or C, for example.

Network architecture

Word2vec architecture was groundbreaking in a sense, that it enabled effective training on much more data compared to the earlier neural network language models. Word2vec tries to minimize the complexity of the preceding models (like the one in Figure 10) by removing the non-linear hidden layer. Only one hidden layer remains, and it has no activation function. On the output layer the model has softmax as an activation function, in other words, as a classifier. The architecture is illustrated in Figure 11.

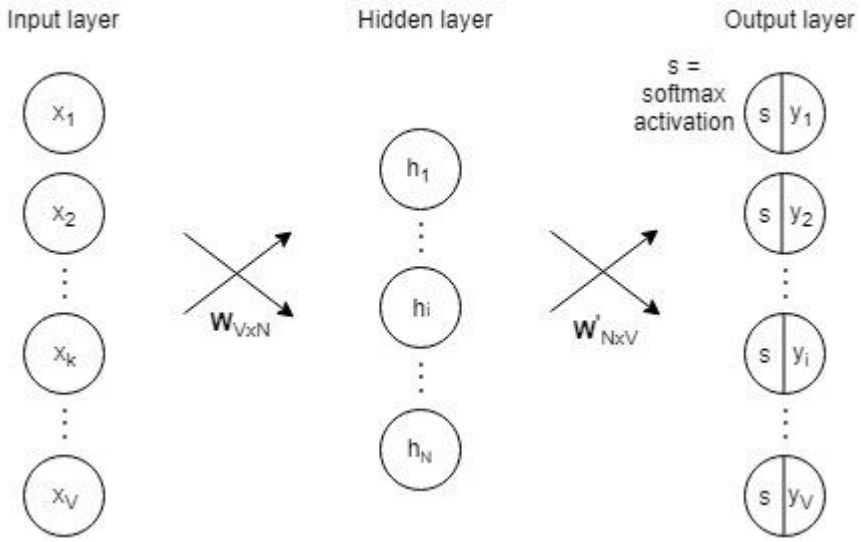

Figure 11. *Word2vec CBOW bigram model predicting the next word (output) given the current word (input) (Rong, 2014, p. 2).*

The architecture is based on a model Mikolov et al. (2009) presented in a conference article considering neural network based language models for highly inflected languages. Finnish is one of these. Hence, word2vec is very interesting from our perspective.

Training the model

The idea of word2vec is easiest to understand explained by CBOW model and bigrams (pairs of sequential words). The model visualized in Figure 11. We can illustrate this with

a simple example. We have a training data of one sentence: "*My pet cat meows*". Hence, our four-word vocabulary is {*my*, *pet*, *cat*, *meows*} and we will denote its size as $V = 4$. The data have three bigrams: "*my pet*", "*pet cat*" and "*cat meows*". Let us choose the word *cat* as our input word and $N = 3$ as our hidden layer size, which will produce us 3-dimensional word embeddings. Input words are fed into the network as one-hot-vectors of size V. Since *cat* is the 3rd word in our vocabulary, its one-hot representation is $[0\ 0\ 1\ 0]^{\mathrm{T}}$. The hidden layer representation for *cat* will be formed in a following way

$$\boldsymbol{h}^{cat} = \boldsymbol{W}^{\mathrm{T}}\boldsymbol{x} = \begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \\ W_{41} & W_{42} & W_{43} \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} W_{31} \\ W_{32} \\ W_{33} \end{bmatrix}$$

The resulting vector is the transpose of the 3rd row of the weight matrix $\boldsymbol{W}$. Likewise, in general case each row $\boldsymbol{W}_{k:}$ of the weight matrix $\boldsymbol{W}$ (each column $\boldsymbol{W}^{\mathrm{T}}{}_{*,k}$ of its transpose $\boldsymbol{W}^{\mathrm{T}}$) gives the N-dimensional hidden layer representation for the given input word w_k

$$\boldsymbol{h} = \boldsymbol{W}^{T}\boldsymbol{x} = \boldsymbol{W}^{T}{}_{*,k}\,\boldsymbol{x} = \boldsymbol{W}_{k,*}\,\boldsymbol{x}.$$

The purpose of the hidden layer is to ensure that words predicting similar probability distribution will also share some of this hidden representation (Mikolov *et al.*, 2009). When the training is complete, the final weights $\boldsymbol{W}_{k,}*$ will construct the feature vector for the input word w_k, i.e., its word embedding.

In Figure 11, weight matrix $\boldsymbol{W'}$ maps the hidden layer representation to the output layer. In our example

$$y^{cat} = \boldsymbol{W'}^{\mathrm{T}}\boldsymbol{h}^{cat} = \begin{bmatrix} W'_{11} & W'_{12} & W'_{13} & W'_{14} \\ W'_{21} & W'_{22} & W'_{23} & W'_{24} \\ W'_{31} & W'_{32} & W'_{33} & W'_{34} \end{bmatrix}^{\mathrm{T}} \boldsymbol{h}^{cat} = \begin{bmatrix} \boldsymbol{W'}^{T}{}_{*,1}\,\boldsymbol{h}^{cat} \\ \boldsymbol{W'}^{T}{}_{*,2}\,\boldsymbol{h}^{cat} \\ \boldsymbol{W'}^{T}{}_{*,3}\,\boldsymbol{h}^{cat} \\ \boldsymbol{W'}^{T}{}_{*,4}\,\boldsymbol{h}^{cat} \end{bmatrix}.$$

The vector on the right gives a score for every word in our 4-word vocabulary. The first row of the weight matrix $\boldsymbol{W'}$ (the first column of its transpose $\boldsymbol{W'}^{\mathrm{T}}$) contributes to the score of the 1st word in our vocabulary, and other columns do respectively. In general, we can denote

$$score_{w_j} = \boldsymbol{W'}_{j,*}\,\mathbf{h} = \boldsymbol{W'}^{T}{}_{*,j}\,\mathbf{h},$$

where w_j is the j^{th} word in our vocabulary. With random initial weights these are just random scores. However, our objective is to get the scores to represent the probability distribution for the next words (denoted y_j in Figure 11) given the input word. In other words, all the scores must be squashed between 0 and 1. Softmax obtains a multinomial distribution for the next word probabilities as follows

$$y_j = p(w_j|w_{input}) = \frac{\exp(score_{w_j})}{\sum_{j\prime=1}^{V} \exp(score_{w_{j\prime}})} = \frac{\exp(\boldsymbol{W'}^T_{:j}\,\mathbf{h})}{\sum_{j\prime=1}^{V} \exp(\boldsymbol{W'}^T_{:j\prime}\,\mathbf{h})},$$

where y_j is the output of the j^{th} neuron in the output layer (Rong, 2014).

The network gives a probability distribution vector $\boldsymbol{y} = [y_1 \ \dots \ y_V]^T$ as an output. Our ground truth (the "actual output") is the desired one-hot vector $\boldsymbol{d} = [0 \ \dots \ 1 \ \dots \ 0]^T$ calculated directly from the input data, with the value of the most probable next word given the input word as 1 and all other elements as 0. In our example we had only one sentence, "*my pet cat meows",* as our training data, hence the probabilities for the next word given the input word *cat* are: *p*(*my*|*cat*) = 0, *p*(*pet*|*cat*) = 0, *p*(*cat*|*cat*) = 0 and *p*(*meows*|*cat*) = 1. That gives us the desired vector $\boldsymbol{d} = [0\ 0\ 0\ 1]^T$ for the input word *cat*.

According to Rong (2014) word2vec maximizes the probability of predicting the desired output word w_{j*} given the input word *x* by minimizing its negative log-likelihood. Hence, the loss function used is

$$L = -\log p(w_{j*}|x) = -(score_{w_{j*}} - \log \sum_{j\prime=1}^{V} \exp(score_{w_{j\prime}}))$$

$$= \log \sum_{j\prime=1}^{V} \exp(score_{w_{j\prime}}) - score_{w_{j*}},$$

where *j** is the index of the desired output word. The loss function *L* is back propagated through the network using standard back propagation algorithm (the idea is explained in Chapter 3.5.3). Weights of the hidden layer are updated accordingly. After each training epoch, the network computes the probability for the test data and if it does not improve enough, the learning rate is halved. No generalization such as weight decay is needed, since it has been noticed to give only slight improvement of the result (Mikolov *et al.*, 2009).

Training can be speeded up by merging very rare words into one symbol. This should be done for another reason also: extremely rare words do not have enough training examples to be mapped as meaningful word vectors and they could only confuse the model. In the experiments of Mikolov et al. (2009) merging all words occurring less than five times reduced the vocabulary size and hence the training time to 25 % of the original.

CBOW bigram model predicting the next word given the current word is the simplest implementation of word2vec but lacks in performance compared to more complicated implementations. CBOW n-gram model predicts the current word given the surrounding words (e.g., 2 preceding words and 2 following words, word order is insignificant). Skip-gram, on the contrary, predicts the surrounding words given the current word (e.g., 2 preceding words and 2 following words in the right order). The idea of the model architectures is illustrated in Figure 12. Model implementation and learning, especially skip-

gram model with negative sampling (sampling the words not among the desired context words), can be studied in more detail from Mikolov et al. (2013).

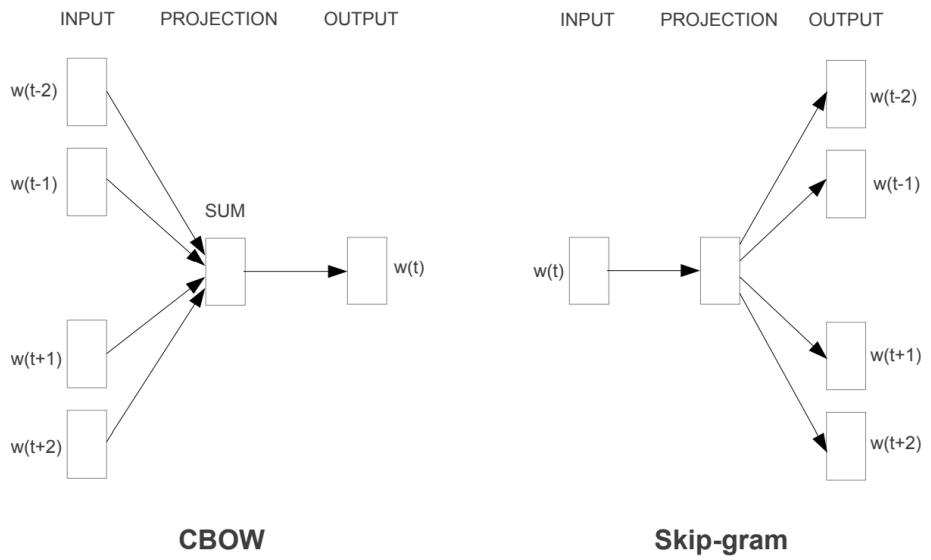

Figure 12. *Word2vec CBOW and skip-gram models with four word context windows (Mikolov, Corrado, et al., 2013, p. 5)*

If the hidden layer projection includes 300 neurons, these models give us a 300-dimensional word embedding for every input word. Each of these dimensions represent some latent feature of the word. Features can be morphological, syntactic or semantic. Some of them (or their combinations) might be quite understandable for us, for example, how animal-like the word is or is it plural or singular. Some features might not be interpretable at all.

Word2vec algebra

Calculating the analogies between words using word2vec embeddings is extremely interesting. Church (2017) considers this an important reason for the popularity of the model. The algebraic analogy hook is promising enough and encourages the NLP community to find new ways to exploit it and improve it.

Word2vec, as other word embeddings, measure the similarity between words by how close their embedding vectors are to one another. The distance is measured by cosine similarity (Mikolov, Corrado, *et al.*, 2013; Mikolov, Yih and Zweig, 2013). In other words, the angle between the word feature vectors (all normalized to unit norm) determine the similarity between the words (see the cosine similarity formula in Chapter 3.5.2). The most used word analogy task is to give an example word and find its most similar word(s), like *cat* is similar to *dog* and other furry 4-legged animals. However, we do not have to stick to these simple similarities with word2vec. Cosine similarity is very effective in solving also more complex similarity tasks by algebraic equations. For example, we can

solve the question "What word is similar to *king* like *man* is to *woman*?" In algebraic form this can be expressed as follows

$$\boldsymbol{man} - \boldsymbol{king} = \boldsymbol{woman} - \boldsymbol{x}$$

$$\boldsymbol{x} = \boldsymbol{king} - \boldsymbol{man} + \boldsymbol{woman},$$

where all the elements are word vectors. It is rather impossible that vector $\boldsymbol{x}$ was exactly the embedding of a particular word in our corpus. Hence, we need to find the word embedding closest to $\boldsymbol{x}$ according to cosine similarity. i.e., the embedding $\boldsymbol{x'}$ with the highest

$$\cos(\boldsymbol{king} - \boldsymbol{man} + \boldsymbol{woman}, \boldsymbol{x}').$$

The result will be $\boldsymbol{x'} = \boldsymbol{queen}$. This vector shift is illustrated in Figure 13. The word embedding succeeds in capturing the gender of the word, one of the word features.

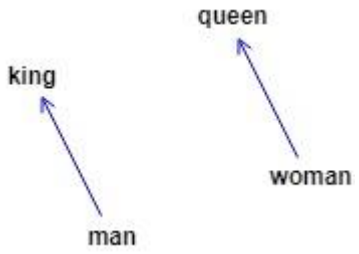

Figure 13. *Vector shift (relation) between man and king is the same as between woman and queen (Mikolov, Yih and Zweig, 2013, p. 749)*

According to Mikolov et al. (2013), same algebra can be applied to many other kinds of analogies between words: semantic, syntactic and morphologic. Gallay and Simko (2016) used it in a fascinating context, namely to obtain the lemma for the word from its inflection. The approach is similar to the gender relation above. The question is now "What word is similar to *happines* like *ill* is to *illness*?" giving us the equation

$$\boldsymbol{illness} - \boldsymbol{ill} = \boldsymbol{happiness} - \boldsymbol{x}$$

$$\boldsymbol{x} = \boldsymbol{illness} - \boldsymbol{ill} + \boldsymbol{happiness}.$$

Again, probably there are several word vectors near the direction of the expected lemma vector $\boldsymbol{x}$. These are all lemma candidates, which are illustrated as a dotted circle in Figure 14. The correct lemma of all candidates must have certain connection with the input word *happiness* based on common letters and other morphological factors.

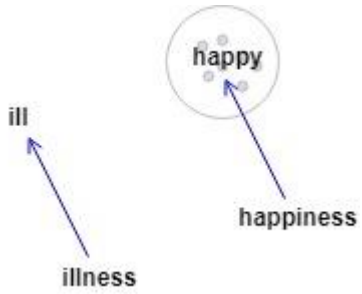

Figure 14. *The expected lemma is in the surroundings of the vector shift, hence all candidates must be investigated (Gallay and Šimko, 2016, p. 535).*

The popularity of word2vec in NLP tasks can best be realized by doing a literature search by the keyword *word2vec* in some scientific search engine or simply by googling. Trained word2vec embeddings are available on the web for numerous languages, for example Finnish[16]. Another highly referenced modern word embedding technique GloVe[17] is also worth exploring while considering corpus-based word similarity measuring.

3.5.5 FastText character n-gram embeddings

Up to this point, all introduced word feature vectors have based solely on word occurrences in corpus. However, there are no words without letters. Bojanowski et al. (2016) introduced *fastText* to tackle the limitation of word level models that assign a distinct vector to each inflection of the word. In fact, fastText is word2vec extended with subword information. It learns distributed vector representations for character n-grams and each word is represented as the sum of its n-gram vectors. Actually, already Schütze (1993) tried slightly similar approach with 4-grams and SVD.

FastText model is derived from continuous skip-gram model introduced by Mikolov et al. (2013). The illustration of the model can be seen in Figure 12. Each word is supplemented with < as a beginning symbol and > as an end symbol. Hence, in case $n = 3$ we can present the word *wheel* by character 3-grams as

```
<wh, whe, hee, eel, el>
```

In addition, the word itself is always included in the set of n-grams (no matter how long it is) to learn a representation for it as well

```
<wheel>
```

Note, that the beginning/end-notation helps us distinguish `eel, the` subword of *wheel*, from the word *eel*, which as a full word is presented as a 5-gram `<eel>`.

[16] Turku BioNLP Group Finnish Internet Parsebank: http://bionlp.utu.fi/finnish-internet-parsebank.html
[17] GLObal VEctors for word presentation: https://nlp.stanford.edu/projects/glove/

According to Bojanowski et al. (2016) the practical approach uses extraction of all 3-grams, 4-grams, 5-grams and 6-grams from the words. If we have a dictionary of size G of these character n-grams g, the given input word x is constructed of an n-gram-set $G_x \subset \{g_1, g_2, \ldots, g_G\}$, where subscripts 1, 2, …, G denote the indices of the n-grams in the dictionary. Each n-gram g is associated with a vector representation $\boldsymbol{g}$. The network tries to predict the desired context word d (vector $\boldsymbol{d}$) for the input word x (vector $\boldsymbol{x}$). The input word is now represented as a sum of all its n-grams. Hence, the scoring function for the given input word x and a desired context word d can be given as follows

$$score(x, d) = \sum_{g \in G_x} \boldsymbol{g}^T \boldsymbol{d}.$$

While learning the right surrounding context word(s) d the network will learn a distributed vector representation for every character n-gram. Using these character n-gram embeddings, word representations can be computed for words not appearing in the training data (Bojanowski *et al.*, 2016).

There exist many other interesting approaches that use embeddings for different kinds of NLP tasks. For example, character embeddings are used in text normalization (Chrupała, 2014) and POS-tagging (Ling *et al.*, 2015). Embeddings can even be constructed for relations between words in an ontology (Goodfellow, Bengio and Courville, 2017).

4. PRACTICAL IMPLEMENTATION: HARNESSING AI TO EXTRACT SKILLS FROM TEXT DATA

The introduced practical AI implementation is empowered by a Finnish AI company Headai[18], the enterprise that commissioned the practical part of the thesis. The aim of the implementation is to solve the complex concept matching problem. The application is mostly considered as a black box and viewed only in terms of its input (text data) and output (skills) without diving deeper into its internal details or algorithms. However, some observations will be considered.

4.1 Input data

The two principal data domains used in the setting of this thesis were curriculum data from higher education institutions and job data from the labor market.

4.1.1 Curriculum data resources

The thesis writer's contribution to the application was programming a web scraper for the curriculum data. The data were scraped from three Finnish Universities of Applied Sciences: Laurea, Metropolia and Haaga-Helia. Together these institutions form a co-operation partnership called The Helsinki Metropolitan Universities of Applied Sciences, abbreviated as 3UAS (3AMK in Finnish). According to Moiso (2018), the partnership has specific strategic co-operation areas, such as education export, research & development, student mobility, innovation, business co-operation and entrepreneurship. However, each institution has its unique educational profile and identity, and strengthening this is also an important objective of the partnership. The distinctive profiles combined with co-operation in focus areas strive to respond to Helsinki metropolitan area's future skills demand. One of the actions taken is using AI and data driven curriculum development to make the institutions' curriculums meet the labor needs in the best possible way. In addition to identifying possible gaps in skills supply, this can help to detect unnecessary overlaps in curriculums.

The content of the education a school supplies is described in its curriculum. Thus, it acts as a student's tool for planning studies and teacher's tool for planning teaching. The curriculum defines the competence targets and the learning outcomes for a whole degree and the studies it includes. The degree consists of core competence and complementary competence modules, which in turn are implemented as study units (usually called courses)

[18] Headai Ltd web site: http://www.headai.com

or projects. The learning outcomes of a module describe the expertise of the entire competence area and the outcomes of courses and projects have their own, more specific targets. The contents of the curricula are constantly evolving and are reviewed annually. (Kokko, 2018; Laurea, 2018; Metropolia, 2018)

The most valuable and relevant data for the skills extraction are the goals and learning outcomes of the modules, courses and projects in the curricula. They should tell us the competencies students possess after completion. By these competencies we can construct the skills supply of the school. Matching that information against the demands on the job market reveals whether the supply meets the demand: perhaps some relevant competencies needed in the labor market are not sufficiently present in the curricula, or some new skills might lack altogether. On the other hand, some traditional focus areas of the school might not be interesting anymore from the labor market's point of view. In turn, evaluating the curricula of the 3UAS universities against each other, we can spot if there exist any unnecessary overlaps. The universities have a strategic partnership, which aims to strengthen their own distinctive profile. Data driven curriculum development provides a strong basis also for this objective.

By the time of writing this thesis, no API helpful enough to make the information extraction process easier was available. Therefore, web scraping was the best choice for implementation. Two of the three universities followed a coherent structure on their web curricula and one had a more customized solution. Hence, the programmed web scraper took advantage of all three elements of Massimino's (2016) categorization: using embedded identifiers, tree-based navigation and searching for contextual identifiers (see Chapter 2.2.1).

The scraping strategy was to gradually drill down deeper into details of curricula and store all the relevant information on the way. Process began from the school level, continuing via the degree and the degree programme level to the most detailed module and course level. It finally contained the learning outcomes, the main information for our task. From these learning outcomes can the skills be extracted by the help of AI and NLP. To facilitate the job of AI and further processing, all relevant information concerning the modules and courses was stored into the database. In addition to the learning outcomes this involved, for example,

- titles of schools, degree programmes, modules and courses,
- teaching language,
- course and module URLs,
- course-lists for the modules.

URLs are important for interactivity and transparency, when we want to evaluate the output of the application.

It is also possible to enrich the curriculum data with other relevant data resources that describe the focus and profile of the university, such as university strategy. Albeit there is a solid expectation that this information has found its way to curricula already and does not give any additional benefit to our task. Various publications of the staff can be exploited as well to gain knowledge about the skills the schools have potential to supply their students with.

4.1.2 Job data resources

The information describing the demands of the current and future job market was gathered from several job service sites and enriched with additional information from other relevant sources. Headai had already accomplished collecting the job data, so there was no contribution from the thesis writer.

The job service sites for the task were chosen on the basis of their coverage, both internationally and locally. The most significant international service used was Monster. Monster is a global online employment solution for people seeking jobs and employers seeking employees. Besides traditional matching of job seekers with jobs, Monster provides career and talent management and a vast array of related products and services in more than 40 countries. Monster is continually developing its services with intelligent digital, social and mobile solutions, and aims to renew the whole recruiting industry. (Monster, 2018)

The other important service for the task was the Public employment and business services (TE Services), which has been a powerful local actor in the job seeking field in Finland for a long time. Ministry of Economic Affairs and Employment of Finland (2018) is responsible for employment, entrepreneurship and labor policy in the country, and thus directs, steers and monitors how TE Services provide their resources for individual customers, enterprises and organizations. In addition to these two main job services, information has been extracted from other job sites as well.

If the only job data gathered was limited to current job announcements, there could be a risk of the data being not enough future oriented. Of course, the skills listed in announcements reflect the competencies needed in the future, but this information can still be enriched to gain even better results. In this case, the enriching was carried out by using data from several other relevant business-related sites in Finland, including Statistics Finland[19], Business Finland[20] and Business Information Systems (BIS)[21]. By the information extracted from these sources we gain knowledge about, for example, what kind of projects

[19] Statistics Finland is a public authority established for statistics: https://www.stat.fi/org/index_en.html
[20] Business Finland supports and funds Finnish innovations: https://www.businessfinland.fi/en/
[21] BIS finds basic information on all companies that have a Business ID: https://www.ytj.fi/en/

have been funded. This, in turn, tells us what skills will be needed in the future and we can truly emphasize the future focused approach to the demands of the labor market.

4.2 Visualized output: How does skills supply correspond to demand?

The introduced application is based on an AI service called Microcomptencies[22]. Microcompetencies is an expert service currently at its beta version offering diverse and broad skills reports based on public data. The skills supply and demand are visualized into skills maps.

In the case of data driven curriculum development, the service first reads a large number of public curriculum data and an immense amount of public job market data, which have been gathered into databases by web scraping. Then the skills are extracted from this input data using text mining and NLP. Skills are initially just words, and AI must find their meaning to be able to compare their similarity. Same skills are commonly expressed with various words.

Skills clearly have a structure that forms an ontology, for example, *Java* `is_a` *programming language*, *Python* `is_a` *programming language*, *programming language* `belongs_to` *programming*, *programming* `belongs_to` *software engineering*. They are also parts of natural language sentences in texts describing job requirements, curriculum objectives, course contents and encyclopedia entries. So, both knowledge-based and corpus-based approaches can be exploited in computing semantic similarity between words. Microcompetencies service uses a dynamic ontology created by Headai's own AI, which links skills and their validation to internationally recognized standards such as ESCO[23] (European Skills, Competences, Qualifications and Occupations) and O*net[24] (The Occupational Information Network) among other ontologies (Headai, 2018).

Skills are just small fractions of the whole data fed into the application. Regarding the knowledge extraction, most of the input data can be considered as noise. This emphasizes the text preprocessing phase. The usual stop word list of too frequent meaningless words or characters is not enough for text cleaning in this case. Other ways for noise removal must be used, too. This often requires a combination of manual and computational effort. The same applies to the ontologies. Field of skills (and occupations) is constantly evolving, and even though an exhaustive ontology of the domain could be constructed today, it will not remain exhaustive for very long. Continuous evaluation, both manual and computational, must be performed. However, we do not want to take a closer stand on the used methods or guess how they are utilized in this particular application.

[22] Microcompetencies skills report service: https://www.microcompetencies.com
[23] Esco classification: https://ec.europa.eu/esco
[24] O*NET taxonomy: https://www.onetonline.org

The information discovered by text mining and processed by NLP cannot be considered as knowledge until it is understandable by the user. Successful visualization is the key to proper interpretation of the results. Usually this means some effective case specific graphical presentation. At its best, the presentation is interactive, and hence provides means to refine the results as well as to picture the discovered knowledge from different angles and at different conceptual levels (Jambhorkar and Jondhale, 1999).

Microcompetencies service uses semantic skills maps for visualization. According to Ketamo (2009) semantic skills map is inspired by Kohonen's (1982, 1990) Self-Organizing Map (SOM). Skills map creates clusters of related skills locating the most relevant cluster at the bottom left of the map and uses color density coding to express the importance of skills. In Figure 15 we can see the bottom left parts (6x6 cells) of two skills maps. The map on the left is created from the skills supplied by a Finnish University of Applied Science and the map on the right is created from the skills demanded by the job market in the Helsinki region in 2018. The skills in the maps are in Finnish and the values inside the brackets denote the word hits in the data.

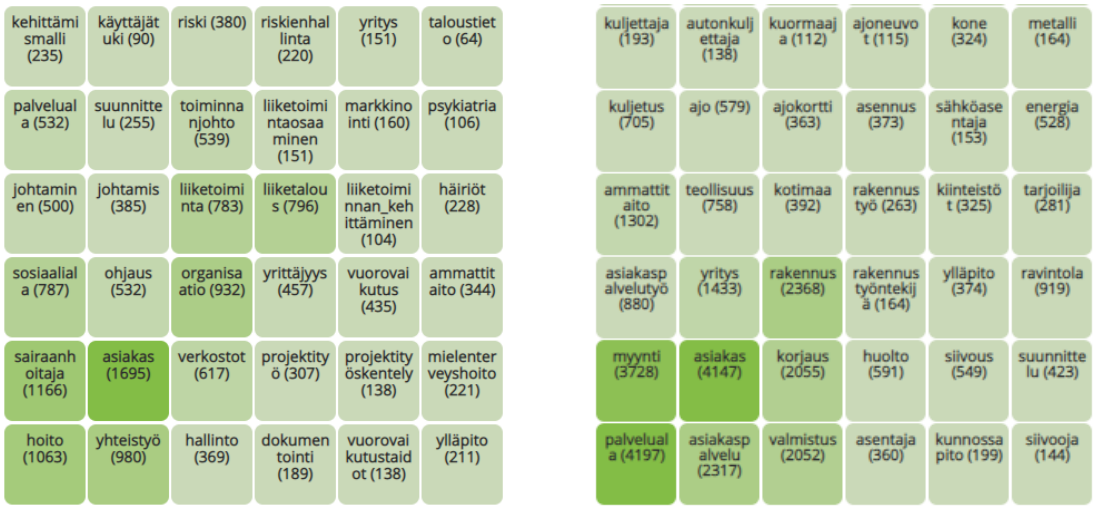

Figure 15. *Parts of semantic skills maps (in Finnish) created from the core skills supplied by a school (left) and the core skills demanded by the job market in Helsinki region (right).*

Figure 16 merges the previous skills maps to visualize how the skills supply meets the demand. The base map shows the skills demanded by the job market, and the color density coding shows how the skills supply of the school matches it. The redder the skill, the deeper the gap in the skills supply of the school. The values inside the brackets denote the combined word hits in the data.

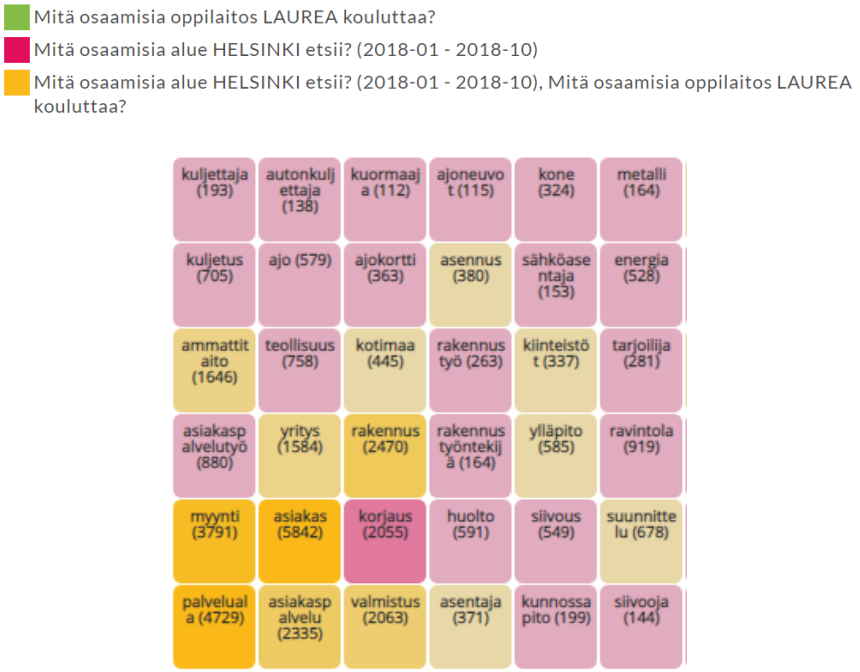

Figure 16. *Merged map of the skills demand in the labor market and how the skills supply of the school responds to it.*

Similar merged map could be created from the skills supplied by two schools and compare their similarities and differences. In this case, the color encoding would tell whether the skill is special for one school or common for both schools.

The 6x6 cell fractions of the maps introduced here demonstrate the output of the application. They concentrate only on the most frequently mentioned skills. The full maps could include hundreds of cells and they show the marginal skills as well. The application learns all the time while processing new data and will be able to create more and more detailed skills maps from the input data drilling deeper into the supplied and demanded micro skills (microcompetencies) on certain fields. Clusters created by different skills could give interesting hints about the interfaces needed in the future curriculums. For example, we might spot a demand for a completely new degree programme combining diverse fields.

5. CONCLUSIONS

During the thesis we pursued answers to three research questions. The first one was to consider the *impact of natural language processing (NLP) in data driven curriculum development*. The role of NLP is crucial for text-related AI tasks, such as data driven curriculum development. Otherwise, the words would be just character strings without meaning for a computer. In fact, the whole thesis dealt with this subject more or less.

We approached the problem by studying the path of data refining step by step from HTML coded content on a web page to knowledge gained via AI application. The initial data was collected by web scraping. Web scraping is one of the most independent methods for gathering public text data to be used as an input in AI tasks. However, data ownership and terms of use must always be taken into consideration while processing web scraping. Scraping data from badly organized or tightly protected web content might sometimes require a great effort. Fortunately, there exist many helpful tools to ease the process. With these tools we can retrieve the HTML code from a website, parse it into an object, and isolate and process the desired data.

To refine our data further, we had to supplement our scope with the second research problem, which reviewed the *processes needed in extracting semantics from words*. Semantics extraction by linguistic computing methods sets some requirements for the input data. These requirements are met by preprocessing the text before semantics extraction. We can use tokenization to split text into smaller, meaningful, text parts, such as words. These so-called tokens are then processed to make text easier to understand for a computer. All NLP tasks benefit from thorough text cleaning, where all redundant information (i.e., noise, like punctuation and irrelevant words) is removed from the text before further processing. In fact, sometimes most of the input text could be considered as noise regarding knowledge extraction.

Another important preprocessing step is text normalization, which finds one unique form to represent all inflections of the word in text. Words could be normalized by stemming or lemmatization. Stemming relies on chopping the suffix and keeping the stem of the word, whereas lemmatization tries to find the basic form of the word (the lemma). Preprocessing is a critical task especially considering morphologically rich languages, such as Finnish. There exists no common framework for preprocessing. The steps taken usually depend on the given NLP task and the language.

To understand natural language, computer must learn to evaluate semantic relatedness between words, which is a critical part of semantics extraction. Semantic similarity is a type of semantic relatedness, that concentrates on taxonomic (hierarchical) likeness of

words. Our third research problem, the major one, concentrated on different *methods semantic relatedness and similarity computation could be performed with*.

The approaches to word similarity computing can mainly be divided into two categories, knowledge-based methods and corpus-based methods. Word representation and similarity computation differ totally between the two methods. Knowledge-based methods rely on ontologies that represent the hierarchical structure of concepts and the relations between them. Concepts are represented as nodes of the directed ontology graph and edges are the relations between them. Knowledge-based methods traverse the ontology graph and measure the path length between the compared concepts. Basically, the shorter the path length, the more similar the concepts are. The methods can also explore the relations between the concepts and the nearest common ancestors of them, as well as their depth in the ontology. In addition, some knowledge-based methods compute the importance of the word according to its frequency in the corpus, i.e., large collection of text.

The actual corpus-based methods, in turn, rely solely on the information retrieved from corpora. They exploit statistical information about the words and their contexts (the surrounding words). Corpus-based methods are mostly based on calculating the context word probabilities for a given word (or vice versa). The most effective modern approaches use neural networks for predicting the context words. During the process the network learns a distributed continuous vector representation for every word, called word embedding. Every dimension of the embedding represents one feature of the word. Word similarity can be measured by the angle between the word embeddings in vector space. The smaller the angle, the more related (and in some cases similar) the words are. Knowledge-based and corpus-based methods can also be combined to gain better results.

However, the ability to give the words a semantic representation and measure the similarity between them is not enough for an AI application. At this stage, the gained knowledge is not visual for a user. A proper full stack AI application is able to visualize the results in a human understandable way. Only then we can talk about true knowledge and wisdom supporting decision making. The practical AI application considered in this thesis used semantic maps for visualization.

While reviewing all these aspects of NLP, the scope of the thesis became quite large. Still, some important subjects, such as word sense disambiguation, had to be left out. Content had to be kept at rather general level. A stricter scope would have made it possible to explore the subject from every relevant perspective and dive deeper into details.

REFERENCES

Alam, S. and Yao, N. (2018) 'The impact of preprocessing steps on the accuracy of machine learning algorithms in sentiment analysis', *Computational and Mathematical Organization Theory*. Springer US, pp. 1–17. doi: 10.1007/s10588-018-9266-8.

Bengio, Y. (2008) 'Neural net language models', *Scholarpedia*, 3(1), p. 3881. doi: 10.4249/scholarpedia.3881.

Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C., Kandola, J., Hofmann, T., Poggio, T. and Shawe-Taylor, J. (2003) 'A Neural Probabilistic Language Model', *Journal of Machine Learning Research*, 3, pp. 1137–1155.

Blumauer, A. (2014) *From Taxonomies over Ontologies to Knowledge Graphs*, *Semantic Web Company*. Available at: https://semantic-web.com/2014/07/15/from-taxonomies-over-ontologies-to-knowledge-graphs/ (Accessed: 12 July 2018).

Bojanowski, P., Grave, E., Joulin, A. and Mikolov, T. (2016) 'Enriching Word Vectors with Subword Information', *Transactions of the Association of Computational Linguistics*, 5(1), pp. 135–146. doi: 1511.09249v1.

Borst, W. N. (1997) *Construction of engineeriing ontologies for knowledge sharing and reuse*, *Technology*. doi: 10.1006/ijhc.1996.0096.

Brychcín, T. and Konopík, M. (2015) 'HPS: High precision stemmer', *Information Processing & Management*. Pergamon, 51(1), pp. 68–91. doi: 10.1016/J.IPM.2014.08.006.

Budanitsky, A. and Hirst, G. (2001) 'Semantic distance in WordNet : An experimental , application-oriented evaluation of five measures', *Workshop on WordNet and Other Lexical Resources, Second meeting of the North American Chapter of the Association for Computational Linguistics*, 2(12), pp. 29–34. doi: 10.1.1.29.2985.

Cambria, E., Schuller, B. B., Xia, Y. and Havasi, C. (2013) 'New Avenues in Opinion Mining and Sentiment Analysis', *IEEE Intelligent Systems*, 28(2), pp. 15–21. doi: 10.1109/MIS.2013.30.

Cammack, R., Atwood, T., Campbell, P., Parish, H., Smith, A., Vella, F. and Stirling, J. (2006) 'Natural language processing', in *Oxford Dictionary of Biochemistry and Molecular Biology*.

Chandler, D. and Munday, R. (2016) *A Dictionary of Social Media*. Oxford University Press. doi: 10.1093/acref/9780191803093.001.0001.

Chen, M., Mao, S. and Liu, Y. (2014) 'Big Data: A Survey', *Mobile Networks and Applications*. Springer US, 19(2), pp. 171–209. doi: 10.1007/s11036-013-0489-0.

Chen, P.-I., Lin, S.-J. and Chu, Y.-C. (2011) 'Using Google latent semantic distance to extract the most relevant information', *Expert Systems with Applications*. Pergamon,

38(6), pp. 7349–7358. doi: 10.1016/J.ESWA.2010.12.092.

Chrupała, G. (2014) ‘Normalizing tweets with edit scripts and recurrent neural embeddings’, in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, pp. 680–686.

Church, K. W. (2017) ‘Emerging Trends: Word2Vec’, *Natural Language Engineering*, 23(1), pp. 155–162. doi: 10.1017/S1351324916000334.

Curran, J. R. (2002) ‘Ensemble methods for automatic thesaurus extraction’, *Proceedings of the ACL-02 conference on Empirical methods in natural language processing - EMNLP ’02*, 10(July), pp. 222–229. doi: 10.3115/1118693.1118722.

Dadashkarimi, J., Nasr Esfahani, H., Faili, H. and Shakery, A. (2016) ‘SS4MCT: A Statistical Stemmer for Morphologically Complex Texts’, in. Springer, Cham, pp. 201–207. doi: 10.1007/978-3-319-44564-9_16.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K. and Harshman, R. (1990) ‘Indexing by latent semantic analysis’, *Journal of the American Society for Information Science*, 41(6), pp. 391–407. doi: 10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9.

Dragoni, M., Da Costa Pereira, C. and Tettamanzi, A. G. B. (2010) ‘An ontological representation of documents and queries for information retrieval systems’, *CEUR Workshop Proceedings*, 560, pp. 83–87. doi: 10.1007/978-3-642-13025-0_57.

Elakiya, E. and Rajkumar, N. (2017) ‘Designing preprocessing framework (ERT) for text mining application’, in *2017 International Conference on IoT and Application (ICIOT)*. IEEE, pp. 1–8. doi: 10.1109/ICIOTA.2017.8073613.

Elliot, M., Fairweather, I., Olsen, W. and Pampaka, M. (2016) ‘Text mining’, in *A Dictionary of Social Research Methods*. doi: 10.1093/acref/9780191816826.013.0152.

European Comission (2018) *2018 reform of EU data protection rules*, *Europa.eu*. Available at: https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en (Accessed: 30 June 2018).

Firth, J. R. (1957) ‘A synopsis of linguistic theory, 1930-1955’, in *Studies in Linguistic Analysis*. Oxford : Blackwell, pp. 1–32.

Friedl, J. E. F. (2002) *Mastering regular expressions*. 2nd ed. O’Reilly.

Gallay, L. and Šimko, M. (2016) ‘Utilizing Vector Models for Automatic Text Lemmatization’, in. Springer, Berlin, Heidelberg, pp. 532–543. doi: 10.1007/978-3-662-49192-8_43.

Glez-Peña, D., Lourenço, A., López-Fernández, H., Reboiro-Jato, M. and Fdez-Riverola, F. (2013) ‘Web scraping technologies in an API world’, *Briefings in Bioinformatics*, 15(5), pp. 788–797. doi: 10.1093/bib/bbt026.

Glorot, X., Bordes, A. and Bengio, Y. (2011) ‘Deep sparse rectifier neural networks’,

AISTATS '11: Proceedings of the 14th International Conference on Artificial Intelligence and Statistics, 15, pp. 315–323. doi: 10.1.1.208.6449.

Goldstone, R. L. (1994) 'Similarity, Interactive Activation, and Mapping', *Journal of Experimental Psychology: Learning, Memory, and Cognition*, pp. 3–28. doi: 10.1037/0278-7393.20.1.3.

Goodfellow, I., Bengio, Y. and Courville, A. (2017) *Deep learning*. MIT Press (Adaptive computation and machine learning series).

Gruber, T. R. (1993) 'A translation approach to portable ontology specifications', *Knowledge Acquisition*. Academic Press, 5(2), pp. 199–220. doi: 10.1006/KNAC.1993.1008.

Guarino, N., Oberle, D. and Staab, S. (2009) 'What Is an Ontology?', in *Handbook on Ontologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–17. doi: 10.1007/978-3-540-92673-3_0.

Han, L., Finin, T. and Joshi, A. (2012) 'Schema-free structured querying of DBpedia data', in *Proceedings of the 21st ACM international conference on Information and knowledge management - CIKM '12*. New York, New York, USA: ACM Press, p. 2090. doi: 10.1145/2396761.2398579.

Harispe, S., Ranwez, S., Janaqi, S. and Montmain, J. (2017) *Semantic Similarity from Natural Language and Ontology Analysis*. preprint. doi: 10.2200/S00639ED1V01Y201504HLT027.

Harris, Z. S. (1954) 'Distributional Structure', WORD, 10(2–3), pp. 146–162. doi: 10.1080/00437956.1954.11659520.

Headai (2018) *Microcompetencies*, *Microcompetencies.com*. Available at: https://www.microcompetencies.com/faq (Accessed: 29 September 2018).

Hedley, J. (2009) *jsoup HTML parser*, *jsoup HTML parser*. Available at: https://jsoup.org/ (Accessed: 11 June 2018).

Henry, N. L. (1974) 'Knowledge Management: A New Concern for Public Administration', *Public Administration Review*, 34(3), pp. 189–196. doi: 10.2307/974902.

Hinton, G., McClelland, J. and Rumelhart, D. (1986) 'Distributed Representations', in *Parallel Distributed Processing*, pp. 77–109. doi: 10.1146/annurev-psych-120710-100344.

Honkela, T. kirjoittaja (2017) *Rauhankone : Tekoälytutkijan testamentti*. Gaudeamus.

Hulpuş, I., Prangnawarat, N. and Hayes, C. (2015) 'Path-based semantic relatedness on linked data and its use to word and entity disambiguation', in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, Cham, pp. 442–457. doi: 10.1007/978-3-319-25007-6_26.

Jambhorkar, S. and Jondhale, V. (1999) *Data Mining Technique: Fundamental Concept*

and Statistical Analysis. Gwalior, India: Horizon Books.

Jiang, J. J. and Conrath, D. W. (1997) 'Semantic Similarity Based on Corpus Statistics and Lexical Taxonomy', *Proceedings of the 10th Research on Computational Linguistics International Conference* , pp. 19–33.

Katz, S. M. (1987) 'Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer', *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3), pp. 400–401. doi: 10.1109/TASSP.1987.1165125.

Ketamo, H. (2009) 'Self-organizing Content Management with Semantic Neural Networks Satakunta University of Applied Sciences', in Mastorakis, N. (ed.) *Recent advances in neural networks : proceedings of the 10th WSEAS International Conference on Neural Networks (NN '09), Prague, Czech Republic, March 23-25, 2009*. Prague: WSEAS, pp. 63–68.

Kettunen, K. (2006) 'Developing an automatic linguistic truncation operator for best-match retrieval of Finnish in inflected word form text database indexes', *Journal of Information Science*. Sage PublicationsSage CA: Thousand Oaks, CA, 32(5), pp. 465–479. doi: 10.1177/0165551506066057.

Kettunen, K., Kunttu, T. and Järvelin, K. (2005) 'To stem or lemmatize a highly inflectional language in a probabilistic IR environment?', *Journal of Documentation*. Emerald Group Publishing Limited, 61(4), pp. 476–496. doi: 10.1108/00220410510607480.

Kimmons, R. (2017) 'Open to all? Nationwide evaluation of high-priority web accessibility considerations among higher education websites', *Journal of Computing in Higher Education*. Springer US, 29(3), pp. 434–450. doi: 10.1007/s12528-017-9151-3.

Kohonen, T. (1982) 'Self-organized formation of topologically correct feature maps', *Biological Cybernetics*, 43(1), pp. 59–69. doi: 10.1007/BF00337288.

Kohonen, T. (1990) 'The Self-Organizing Map', *Proceedings of the IEEE*, pp. 1464–1480. doi: 10.1109/5.58325.

Kokko, T. (2018) *Students' Guide | Haaga-Helia University of Applied Sciences*, *Haaga-Helia University of Applied Science*. Available at: http://www.haaga-helia.fi/en/students-guide (Accessed: 27 June 2018).

Korenius, T., Laurikkala, J., Järvelin, K. and Juhola, M. (2004) 'Stemming and lemmatization in the clustering of finnish text documents', *Proceedings of the Thirteenth ACM conference on Information and knowledge management - CIKM '04*. New York, New York, USA: ACM Press, pp. 625–633. doi: 10.1145/1031171.1031285.

Landauer, T. K., Folt, P. W. and Laham, D. (1998) 'An introduction to latent semantic analysis', *Discourse processes*, 25(2), pp. 259–284. doi: 10.1080/01638539809545028.

Laurea (2018) *Opinto-opas*, *Laurea University of Applied Sciences*. Available at: https://ops.laurea.fi/index.php/en/en (Accessed: 27 June 2018).

Leacock, C. and Chodorow, M. (1998) 'Combining Local Context and WordNet

Similarity for Word Sense Identification', *WordNet: An electronic lexical database.*, (JANUARY 1998), pp. 265–283. doi: citeulike-article-id:1259480.

Li, C., Ma, T., Zhou, Y., Cheng, J. and Xu, B. (2017) 'Measuring Word Semantic Similarity Based on Transferred Vectors', in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, Cham, pp. 326–335. doi: 10.1007/978-3-319-70093-9_34.

Lin, D. (1998) 'An Information-Theoretic Definition of Similarity', *Proceedings of ICML*, pp. 296–304. doi: 10.1.1.55.1832.

Ling, W., Luís, T., Marujo, L., Astudillo, R. F., Amir, S., Dyer, C., Black, A. W. and Trancoso, I. (2015) 'Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation', in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portucal: Association for Computational Linguistics, pp. 1520–1530. doi: 10.18653/v1/D15-1176.

Manning, C. D., Raghavan, P. and Schutze, H. (2009) *An Introduction to Information Retrieval*, *Online*. Cambridge: Cambridge University Press. doi: 10.1109/LPT.2009.2020494.

Manning, C. D. and Schütze, H. (1999) *Foundations of statistical natural language processing*. Cambridge, Mssachusetts: MIT Press.

Massimino, B. (2016) 'Accessing Online Data: Web-Crawling and Information-Scraping Techniques to Automate the Assembly of Research Data', *Journal of Business Logistics*. Wiley-Blackwell, 37(1), pp. 34–42. doi: 10.1111/jbl.12120.

Mayo, M. (2017) *A General Approach to Preprocessing Text Data*, *KDnuggets*. Available at: https://www.kdnuggets.com/2017/12/general-approach-preprocessing-text-data.html (Accessed: 3 July 2018).

McInnes, B. T. and Pedersen, T. (2013) 'Evaluating measures of semantic similarity and relatedness to disambiguate terms in biomedical text', *Journal of Biomedical Informatics*. Academic Press, 46(6), pp. 1116–1124. doi: 10.1016/J.JBI.2013.08.008.

Merriam-Webster.com (2018) *noise*, *Merriam-Webster Online Dictionary*. Available at: https://www.merriam-webster.com/dictionary/noise (Accessed: 6 October 2018).

Metropolia (2018) *Curricula - Metropolia UAS*, *Metropolia University of Applied Sciences*. Available at: http://opinto-opas-ops.metropolia.fi/index.php/en/en (Accessed: 27 June 2018).

Mihalcea, R., Corley, C. and Strapparava, C. (2006) 'Corpus-based and knowledge-based measures of text semantic similarity', *Proceedings of the 21st national conference on Artificial intelligence*, 1, pp. 775–780. doi: 10.1.1.65.3690.

Mikolov, T., Corrado, G., Chen, K. and Dean, J. (2013) 'Efficient Estimation of Word Representations in Vector Space', *Proceedings of the International Conference on Learning Representations (ICLR 2013)*, pp. 1–12. doi: 10.1162/153244303322533223.

Mikolov, T., Kopecky, J., Burget, L., Glembek, O. and Cernocky, J. (2009) 'Neural

network based language models for highly inflective languages', in *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, pp. 4725–4728. doi: 10.1109/ICASSP.2009.4960686.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. and Dean, J. (2013) 'Distributed Representations of Words and Phrases and their Compositionality', in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*. Curran Associates Inc., pp. 3111–3119. doi: 10.1162/jmlr.2003.3.4-5.951.

Mikolov, T., Yih, W.-T. and Zweig, G. (2013) 'Linguistic regularities in continuous space word representations', *Proceedings of NAACL-HLT*. Association for Computational Linguistics, (June), pp. 746–751. doi: 10.3109/10826089109058901.

Ministry of Economic Affairs and Employment of Finland (2018) *Public employment and business services*, *Tem.fi*. Available at: https://tem.fi/en/public-employment-and-business-services (Accessed: 28 June 2018).

Mitchell, R. (2013) *Instant Web Scraping with Java: Build simple scrapers or vast armies of Java-based bots to untangle and capture the Web*. Packt Publishing.

Moisio, A. (2018) *Stronger together - strategic partnership between Haaga-Helia, Laurea and Metropolia | Haaga-Helia University of Applied Sciences*. Available at: http://www.haaga-helia.fi/en/about-haaga-helia/stronger-together-strategic-partnership-between-haaga-helia-laurea-and-metropoli-0?userLang=en (Accessed: 27 June 2018).

Monster (2018) *Our company | Monster.com*, *Monster WorldWide Inc.* Available at: https://www.monster.com/about/our-company (Accessed: 27 June 2018).

Nakov, P., Popova, A. and Mateev, P. (2001) 'Weight functions impact on LSA performance', in *Proceedings of the EuroConference Recent Advances in Natural Language Processing, RANLP 2001*, pp. 187–193.

Patwardhan, S., Banerjee, S. and Pedersen, T. (2003) 'Using Measures of Semantic Relatedness for Word Sense Disambiguation', in. Springer, Berlin, Heidelberg, pp. 241–257. doi: 10.1007/3-540-36456-0_24.

Porter, M. (2014) *Snowball*. Available at: http://snowball.tartarus.org/ (Accessed: 5 July 2018).

Porter, M. F. (1980) 'An algorithm for suffix stripping', *Program*. MCB UP Ltd, 14(3), pp. 130–137. doi: 10.1108/eb046814.

Princeton University (2010) *About WordNet*, *WordNet*. Available at: https://wordnet.princeton.edu/ (Accessed: 13 July 2018).

Rada, R., Mili, H., Bicknell, E. and Blettner, M. (1989) 'Development and application of a metric on semantic nets', *IEEE Transactions on Systems, Man, and Cybernetics*, 19(1), pp. 17–30. doi: 10.1109/21.24528.

Rehman, Z., Anwar, W., Bajwa, U. I., Xuan, W. and Chaoying, Z. (2013) 'Morpheme Matching Based Text Tokenization for a Scarce Resourced Language', *PLoS ONE*. Edited by R. L. Patterson, 8(8), p. e68178. doi: 10.1371/journal.pone.0068178.

Reniers, V., Rafique, A., Van Landuyt, D. and Joosen, W. (2017) 'Object-NoSQL Database Mappers: a benchmark study on the performance overhead', *Journal of Internet Services and Applications*. Springer London, 8(1), p. 1. doi: 10.1186/s13174-016-0052-x.

Resnik, P. (1995) 'Using Information Content to Evaluate Semantic Similarity in a Taxonomy', in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pp. 448–453.

Rong, X. (2014) 'word2vec Parameter Learning Explained', *arXiv:1411.2738 [cs.CL]*.

Rowley, J. (2007) 'The wisdom hierarchy: representations of the DIKW hierarchy', *Journal of Information Science*. Sage PublicationsSage CA: Thousand Oaks, CA, 33(2), pp. 163–180. doi: 10.1177/0165551506070706.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986) 'Learning representations by back-propagating errors', *Nature*. Nature Publishing Group, 323(6088), pp. 533–536. doi: 10.1038/323533a0.

Sadalage, P. and Fowler, M. (2012) *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, *Vasa*. doi: 0321826620.

Saharia, N., Konwar, K. M., Sharma, U. and Kalita, J. K. (2013) 'An Improved Stemming Approach Using HMM for a Highly Inflectional Language', in. Springer, Berlin, Heidelberg, pp. 164–173. doi: 10.1007/978-3-642-37247-6_14.

Sahlgren, M. (2008) 'The distributional hypothesis', *Italian Journal of Linguistics*, 20(1), pp. 33–54.

Salas, J. and Domingo-Ferrer, J. (2018) 'Some Basics on Privacy Techniques, Anonymization and their Big Data Challenges', *Math.Comput.Sci*. Springer International Publishing, pp. 1–12. doi: 10.1007/s11786-018-0344-6.

Salton, G. and McGill, M. J. (1983) 'Introduction to modern information retrieval.', *Introduction to modern information retrieval*, p. 400.

Salton, G., Wong, A. and Yang, C. S. (1975) 'A vector space model for automatic indexing', *Communications of the ACM*. ACM, 18(11), pp. 613–620. doi: 10.1145/361219.361220.

Sánchez, D., Batet, M., Isern, D. and Valls, A. (2012) 'Ontology-based semantic similarity: A new feature-based approach', *Expert Systems with Applications*. Pergamon, 39(9), pp. 7718–7728. doi: 10.1016/j.eswa.2012.01.082.

Scarpetta, S., Sonnet, A., Livanos, I., Núñez, I., Craig Riddell, W., Song, X. and Maselli, I. (2012) 'Challenges facing European labour markets: Is a skill upgrade the appropriate instrument?', *Intereconomics*, 47(1), pp. 4–30. doi: 10.1007/s10272-012-0402-2.

Schuhmacher, M. and Ponzetto, S. P. (2014) 'Knowledge-based graph document modeling', in *Proceedings of the 7th ACM international conference on Web search and data mining - WSDM '14*. New York, New York, USA: ACM Press, pp. 543–552. doi: 10.1145/2556195.2556250.

Schutze, H. (1993) ‘Word Space’, *Advances in Neural Information Processing Systems 5*, 5, pp. 895–902.

Shannon, C. E. (1948) ‘A mathematical theory of communication’, *The Bell System Technical Journal*, 27, pp. 379–423. doi: 10.1145/584091.584093.

Stebbins, R. A. (2008) ‘Exploratory Research’, in *The Sage Encyclopedia of Qualitative Research Methods*, pp. 328–330. doi: 10.4135/9781412963909.

Studer, R., Benjamins, V. R. and Fensel, D. (1998) ‘Knowledge engineering: Principles and methods’, *Data & Knowledge Engineering*. North-Holland, 25(1–2), pp. 161–197. doi: 10.1016/S0169-023X(97)00056-6.

Technopedia (2013) *What is Web Scraping? - Definition from Techopedia*, *technopedia.com*. Available at: https://www.techopedia.com/definition/5212/web-scraping (Accessed: 10 June 2018).

Tversky, A. (1977) ‘Features of similarity’, *Psychological Review*, 84(4), pp. 327–352. doi: 10.1037/0033-295X.84.4.327.

Twitter Inc. (2012) *Twitter Terms of Service*, *Twitter*. Available at: https://twitter.com/tos/ (Accessed: 15 June 2018).

Uysal, A. K. and Gunal, S. (2014) ‘The impact of preprocessing on text classification’, *Information Processing & Management*. Pergamon, 50(1), pp. 104–112. doi: 10.1016/J.IPM.2013.08.006.

W3schools (2018) *JavaScript HTML DOM*, *w3schools.com*. Available at: https://www.w3schools.com/Js/js_htmldom.asp (Accessed: 9 June 2018).

W3Schools (2018) *HTTP Methods GET vs POST*, *w3schools.com*. Available at: https://www.w3schools.com/tags/ref_httpmethods.asp (Accessed: 11 June 2018).

Wang, R., Liu, W. and McDonald, C. (2014) ‘How preprocessing affects unsupervised keyphrase extraction’, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, Berlin, Heidelberg, pp. 163–176. doi: 10.1007/978-3-642-54906-9_14.

Wu, Z. and Palmer, M. (1994) ‘Verb Semantics and Lexical Selection’, in *Proceedings of ACL 94*.

Xiong, H., Pandey, G., Steinbach, M. and Kumar, V. (2006) ‘Enhancing data analysis with noise removal’, *IEEE Transactions on Knowledge and Data Engineering*, 18(3), pp. 304–319. doi: 10.1109/TKDE.2006.46.

Yuhua Li, Bandar, Z. A. and McLean, D. (2003) ‘An approach for measuring semantic similarity between words using multiple information sources’, *IEEE Transactions on Knowledge and Data Engineering*, 15(4), pp. 871–882. doi: 10.1109/TKDE.2003.1209005.

Zhu, G. and Iglesias, C. A. (2017) ‘Computing Semantic Similarity of Concepts in Knowledge Graphs’, *IEEE Transactions on Knowledge and Data Engineering*, 29(1), pp.

72–85. doi: 10.1109/TKDE.2016.2610428.