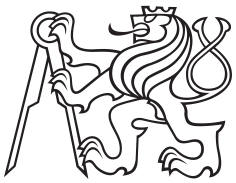
Master's Thesis



Czech
Technical
University
in Prague

FEL

Faculty of Electrical Engineering
Department of Computer Science

Creating a Knowledge Base from Websites

Bc. Josef Štěřovský

Supervisor: Ing. Radek Mařík, CSc.
Study program: Open Informatics
Specialization: Data Science
May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Štěřovský Josef** Personal ID number: **484037**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Data Science**

II. Master’s thesis details

Master’s thesis title in English:

Creating a Knowledge Base from Websites

Master’s thesis title in Czech:

Tvorba znalostní báze z webových stránek

Guidelines:

1/ Create an overview of the methods needed to retrieve the content of company and institutional websites and extract the information contained.
2/ Select appropriate methods and implement the experimental environment.
3/ Scrape a given set of web page references, create an input data corpus from the downloaded web content. Identify required entities and save their attributes and relations into a suitable form of knowledge base. Demonstrate how such a database might be used.
4/ Discuss the obtained results also with respect to their accuracy.

Bibliography / sources:

1/ Hobson Lane, Cole Howard, Hannes Max Hapke: Natural Language Processing in Action, Manning, 2019.
2/ Thushan Ganegedara: Natural Language Processing with Tensorflow, Packt, 2018.
3/ Li Deng, Yang Liu: Deep Learning in Natural Language Processing, Springer, 2018.
4/ Ryan Mitchell. Web Scraping with Python, 2e. Sebastopol, CA: O’Reilly Media, 2018.

Name and workplace of master’s thesis supervisor:

Ing. Radek Mařík, CSc. Department of Telecommunications Engineering FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **01.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Radek Mařík, CSc. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I would like to thank my supervisor, Radek Mařík, for making this thesis possible, for sharing his resourcefulness and experience in advice and stories, and for being genuinely interested in the experiments and providing valuable feedback.

I would also like to thank Adam Dobiáš for supporting the project and for the permission to use the production data labels which were used to assess the quality and accuracy of the models throughout this thesis.

Last but not least, thanks go to my family, friends and colleagues, to everyone who has supported me, and to everyone who has been patient with me for the last year.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 24. May 2024

Abstract

The aim of this thesis is to assess the feasibility of automating knowledge extraction from websites, primarily in classifying businesses into pre-defined categories based on industry sectors. Custom HTML datasets are compiled directly from the websites of businesses operating in several target countries by means of web scraping. Natural language corpora are extracted from the HTML datasets in the English, German and Spanish languages. The corpora are then embedded into vector spaces using large pre-trained models, as well as smaller models trained directly on the datasets. The degree of correspondence between ground-truth categories of the websites and the position of the documents in the vector spaces is visualized using dimension reduction methods. The quality of selected datasets and text extraction tools is also evaluated using the precision of classifiers trained on the labelled document vectors. Alternative strategies are explored as well, namely clustering of the document vectors, prompting LLMs with the extracted documents, and the possibility of using existing open knowledge bases. The creation and querying of a new knowledge base using the predictions of the models and the used category hierarchy is also demonstrated. Ensuring the quality of the corpora is a major issue, training or at least fine-tuning models for the task is very beneficial.

Keywords: knowledge base, text extraction, text classification, web scraping, Scrapy, Doc2Vec, Top2Vec, PaCMAP, Mistral LLM, RDF

Supervisor: Ing. Radek Mařík, CSc.

Abstrakt

Cílem této práce je posoudit proveditelnost automatizace extrakce znalostí z webových stránek, především při klasifikaci obchodníků do kategorií, které jsou předem definovány na základě průmyslových odvětví. Z webových stránek obchodníků působících v několika cílových zemích jsou sestaveny HTML datasety pomocí web scrapingu. Dále jsou pak z HTML datasetů extrahovány korpusy v angličtině, němčině a španělštině. Korpusy jsou poté transformovány do vektorových prostorů pomocí velkých předtrénovaných modelů i menších modelů trénovaných přímo na datasetech. Míra shody mezi skutečnými kategoriemi webových stránek a polohou dokumentů ve vektorových prostorech je vizualizována pomocí metod redukce dimenze. Kvalita vybraných datasetů a nástrojů pro extrakci textu je hodnocena také pomocí přesnosti klasifikátorů natrénových na vektorech anotovaných dokumentů. Rovněž jsou zkoumány alternativní strategie, konkrétně shlukování vektorů dokumentů, LLM prompting pomocí extrahovaných dokumentů a možnost využití již existujících otevřených znalostních bází. Je demonstrováno vytvoření a dotazování nové znalostní báze s využitím předpovědí modelů a použité hierarchie kategorií. Důležitým problémem je zajištění kvality korpusů, velmi přínosné je trénování nebo alespoň fine-tuning modelů pro tuto úlohu.

Klíčová slova: znalostní báze, extrakce textu, klasifikace textu, web scraping, Scrapy, Doc2Vec, Top2Vec, PaCMAP, Mistral LLM, RDF

Překlad názvu: Tvorba znalostní báze z webových stránek

Contents

Figures

Tables

Chapter 1

Introduction

This thesis deals with the extraction of knowledge from websites, using the text content of the websites and machine learning techniques. The use case for this task is to aid in the annotation of businesses for TapiX[1], an API for payment data enrichment used by mobile apps of banking institutions.

The main part of the annotation process that we will investigate is the categorization of the businesses. Card transactions have their own category system of Merchant Category Codes (MCCs). However, there are several issues with MCCs – namely, they are sometimes very broad, very narrow, outdated, or used incorrectly by card terminal operators[2]. For this reason, TapiX uses a custom category system that has a hierarchical, tree-like structure. Due to the unreliability of MCCs, correctness of the categories has to be verified by human specialists.

The correct category labels have been kindly provided by the TapiX data team and they will be used as ground truth throughout this thesis. Links to the websites of the businesses are also available. However, the content of the linked websites is not maintained by TapiX. For this reason, we will need to start by building a dataset from the provided websites. However, we will not be able to publish the created datasets, as they consist of scraped content of business websites, which we have no permission to reproduce.

After processing the data, we will show how to create a knowledge base using the obtained data and the utilized category tree. However, this knowledge base has no affiliation with TapiX and serves only for demonstration purposes.

Below is an outline of the structure of the thesis.

- Chapter 2 is concerned with definitions and theoretical concepts that we expect to be useful when modeling and classifying text data.
- Chapter 3 introduces examples of software and pre-trained models that are freely available and could be helpful for some part of the knowledge extraction process.
- Chapter 4 elaborates on the categorization problem, and proposes strategies to address it.

[1] https://www.tapix.io/

[2] See this article for more details: https://www.tapix.io/resources/post/why-mcc-codes-do-not-help-much-with-payment-categorization

- Chapter 5 describes the implemented experiments and their results.

Attempts have already been made to deal with the categorization problem in [Bor23], focusing on active learning. This thesis will approach the problem more generally, gather more data and utilize various language models which have not been used previously.

Chapter 2

Theoretical foundations

This chapter shall encompass a brief theoretical introduction to the relevant scientific fields and algorithms that might be used during the implementation. Many of the topics described here would probably warrant a full thesis-length text if described properly, so we will focus on introducing the most common and relevant terminology, and trying to provide at least basic intuition to the functioning of the algorithms.

Before we get into specifics, let us first clarify what is the ultimate aim of the thesis, i.e. creating a knowledge base, for there are varying definitions of the term "knowledge base". Some use the term to refer to software systems not entirely unlike databases, but with more focus on capturing the complexity arising from the relationships and hierarchies formed by the described objects, and sometimes also the ability to automatically draw some conclusions that aren't explicitly in the data, but are a logical consequence of the properties of the data. Many others know "knowledge bases" as a different kind of software, facilitating the writing, organizing and sharing of knowledge among people by means of documentation, primarily in a corporate environment, like Atlassian's Confluence[1]. In this thesis, when referring to knowledge bases, we shall follow the former definition, i.e. our knowledge base should be machine-readable and similar to a database.

In recent times, our definition of a knowledge base is associated with the idea of the *Semantic Web* [Blu20]. Many currently active projects describing themselves as knowledge bases according to this definition utilize technologies originally developed for the Semantic Web. These technologies, some of which we will touch upon in Subsection 2.1.1, were originally developed in the 2000s to improve upon the existing Internet standards and provide an alternative communication framework. Nowadays, they are mostly used in research projects to organize complex data and build taxonomies, but they are also employed by websites to help web crawlers make sense of their content. However, the Semantic Web has never been adopted universally and many projects don't use its tools despite having very similar goals. We will not

[1] When searching for the term "knowledge base" using Google, it's likely that all results except Wikipedia will use exclusively this definition. Similarly, if we ask ChatGPT what a knowledge base is, it will most likely only refer to this definition. Here is an article listing "knowledge base software" while completely ignoring the first definition: https://medium.com/@HelpLook/the-best-knowledge-base-software-in-2024-e65c9407113f

dwell upon using Semantic Web technologies too much, as much of this thesis is concerned with learning from unstructured data, and this task remains the same regardless of the expected format of the output.

2.1 Knowledge graphs

When describing a knowledge base, especially one containing complex relationships, it's common to employ the already existing mathematical concept of graphs, using the graph's nodes to represent complex objects in the real world, or *entities* [Zen+21]. There are essentially two kinds of elementary statements that we may wish to express about those entities and the structure of the knowledge base:

- Two entities may be in a *relationship*, which will be mapped to an edge between them. For instance, if Prague is the capital city of Czechia, we will connect the entities `Prague` and `Czechia` with an edge that is named `isCapitalCityOf` or similarly.
- Entities and relationships may have *attributes*, characterizing or annotating them in some way. An example is that Prague has 1.3 million inhabitants. The simplest way to do this is to add an attribute, called e.g. `hasInhabitants`, with the value 1.3 million.

In the case of relationships between entities, it's quite straightforward that the relationship should be a (directed) edge between the entity nodes. However, it's not as clear how to represent an attribute, and in fact, attributes are represented differently in the various existing knowledge graph standards. One common approach is to define a new type of node, often called a *literal*, which stores a simple value (like 1.3 million), and attach attributes to entities by defining edges that specify the kind of attribute stored in the literal (like `hasInhabitants`). Another approach is to expose an interface for annotating the entity nodes extensively, meaning that the attributes are stored directly in the entity node [Vuk15].

Knowledge graphs have long been used for *reasoning*, meaning that an algorithm uses logical rules to infer relationships that are not explicitly in the graph, or find contradictions in the graph. Many early artificial intelligence used this kind of reasoning on knowledge graphs extensively, most notably *Cyc*, a research project active from at least 1986 until today [LG90]. More recently, reasoning has been implemented using *ontologies* [Blu20], which could be seen as subgraphs of the knowledge graphs that describe the rules which should be applied. These rules cannot be too complex[2], or else they quickly become computationally infeasible on large graphs. In ontology implementations, *description logics* [Baa+07] are used to define the constraints on the rules which can be created in a particular knowledge graph.

[2] An example of a reasoning rule would be that if a person A is a child of person B, and person B is a child of person C, then person A is the grandchild of person C.

Due to the popularity of relational databases, various standards have been developed for mapping relational data to graph data and describing such a mapping in a formalized way [Jan+20].

2.1.1 RDF and OWL

RDF (Resource Description Framework) is a data model for graph data, where both entities and relationships are treated as *resources*, each resource having a unique identifier (IRI). It is defined by W3C recommendations such as [KC04] and has an extensive framework for modeling complex relationships, different types of attributes, hierarchies of classes, and much more. The graph is stored as a set of *triples*, where each triple consists of a *subject*, *predicate* and *object*. In the context of a triple, the predicate can be interpreted as an edge pointing from the subject to the object, but the predicate is a resource just like the subject and object, meaning that the predicate of a triple can be a subject or object of another triple. Attribute values are stored as a special type of node in the graph (literal), which is then used as the object of the statement expressing the attribute.

There are several data formats available for storing RDF data, e.g. an XML-based format (*RDF/XML*), a JSON-based format (*JSON-LD*) and a format that simply stores the list of defined triples in a plain text format (*N-Triples*). *RDF Turtle* is a format that extends N-Triples with many shortcuts with the aim of making the data more concise and human-readable. For example, we might express the already used examples in this section in Turtle as

```
@prefix ex: <http://example.org/> .

ex:Czechia a ex:Country ;
    ex:hasCapitalCity ex:Prague .

ex:Prague a ex:City ;
    ex:hasInhabitants 1300000 .
```

Knowing already what the conveyed knowledge is supposed to be, it's probably easy to make out which part of the Turtle example is encoding what knowledge. However, Turtle conceals some of the underlying complexity of the RDF model by design. As defined on the first line, the prefix `ex:` is replaced with the example IRI, meaning that `ex:Czechia` becomes `<http://example.org/Czechia>`. The predicate `a` stands for an internal RDF resource, `rdf:type`, with the meaning being apparent from the example (Prague *is a* city).

The Web Ontology Language (OWL) is another W3C recommendation [Mot+08] for defining ontologies that specify rules for RDF graphs. Like RDF, OWL also has various serialization formats. Some of them are more specialized formal logic formats, but OWL can also be expressed as a subset of RDF, meaning that all valid RDF formats are also valid OWL formats.

Once again using the same example, we might want to define a *class* of cities and express a rule that every country has exactly one capital city. This rule can also be written down in RDF Turtle:

```
@prefix ex: <http://example.org/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Country a owl:Class;
    rdfs:subClassOf [
        a                owl:Restriction ;
        owl:onProperty   ex:hasCapitalCity ;
        owl:cardinality  "1"^^xsd:nonNegativeInteger
    ] .
```

This example uses some advanced features of both the RDF model and Turtle syntax (blank nodes, RDF Schema). The key point is that constructing ontologies directly in RDF can be tedious, so specialized tools are often used instead.

Note that OWL implementations use the *open world assumption*, meaning that if the graph contains no triple defining a capital city for a country, we assume that we do not know what the country's capital city is, instead of asserting that the country has no capital city (*closed world assumption*). Therefore, the rule can only be violated by a country having 2 or more capital cities.

RDF graphs can be queried and updated using SPARQL, a specialized query language. It is up to the database implementation to make it possible to query triples generated or inferred by OWL (OWL *entailment*) and other extensions.

2.1.2 Cypher

Cypher is the query language of Neo4j [Vuk15], one of the most popular graph databases. While RDF, OWL and SPARQL originated as standards that were then implemented by various databases, Cypher was designed explicitly for Neo4j and has a much more unified interface, aiming to be more like a standard database language similar to SQL. This means that unlike in RDF, there are no universal identifiers, no explicit ontology support, and updating and querying the database can all be done using a single language.

The following Cypher snippet inserts the already used capital city example into a Neo4j database:

```
CREATE    (prg:CITY {name: "Prague", population: 1300000}),
          (cz:COUNTRY {name: "Czechia"}),
          (cz)-[h:HAS_CAPITAL_CITY]->(prg);
```

We can then query the database for the capital city of Czechia and its population:

```
MATCH     (ci:CITY)<-[h:HAS_CAPITAL_CITY]-(co:COUNTRY)
WHERE     ci.name = "Czechia"
RETURN    ci.name as city_name,
          ci.population as city_population;
```

The type (label) of nodes and relationships is explicitly declared when creating the nodes and relationships in the database. (In this example, nodes have `CITY` and `COUNTRY` types and the relationship is of the type `HAS_CAPITAL_CITY`.) Unlike in RDF, attributes don't have to be declared as separate nodes, since nodes themselves can have attributes declared using syntax similar to a JavaScript object, as is evident from the example.

2.2 Natural language processing

Natural language processing, or NLP, is an area of computer science concerned with analyzing and generating text in natural languages as used in human-to-human communication. NLP is closely related to computational linguistics (CL) – it might be described either as an applied-science counterpart to CL, or as a subfield of CL when using a broader definition [Tsu21].

With respect to formal language theory, natural languages can prove difficult to pin down, and various extensions to the Chomsky hierarchy have been proposed to account for them [JR12]. Much of NLP research has a more probabilistic than formal nature, though, and is concerned with tasks like language modeling and translation between different languages. The quality of the models can be evaluated either *intrinsically* using task-independent probabilistic metrics, or *extrinsically* by comparing the model's outputs in a particular task to a standard (human) output [JM23]. Since language modeling will only be used in this thesis as a means to the end of generating knowledge, we will not go into detail on these topics. Instead, a couple techniques and basic concepts which are traditionally used in NLP and relevant to our purposes will be introduced here, and some language models will be mentioned later in different contexts.

2.2.1 Tokenization

Tokenization refers to splitting text in natural language into small pieces which are then treated as "units" of information.

An obvious choice is to tokenize the text into words. In simplified terms, a word is the smallest grammatically independent unit of language [Fas08]. This definition is not sufficient in theoretical linguistics, i.a. because words can be further divided into morphemes and it can be ambiguous which morphemes carry meaning by themselves. In many languages, we may define a word as a unit of text separated by whitespace from other words, but some languages don't use spaces to separate words (e.g. Mandarin Chinese), or use spaces to separate syllables even inside words (e.g. Vietnamese).

Even when we decide to use words as tokens and words can be parsed easily in our target language, there are extra considerations such as whether

to lemmatize the words (essentially remove prefixes/suffixes that only serve a grammatical purpose) and normalize them (e.g. convert all characters into lowercase) [JM23].

However, more advanced language models typically use tokens which are smaller than a word, using algorithms such as byte-pair encoding, which essentially compresses the text, using tokens either for sequences which are common in the text, or individual characters when no common sequence is found.

2.2.2 TF-IDF

TF-IDF is a technique for analyzing natural language *corpora* (collections of documents). It converts a corpus into a sparse matrix of scores, where the matrix's two indices represent words and documents, respectively. It is a traditional alternative to deep learning techniques (see 2.6) which is still very popular. According to some recently published research, TF-IDF can still be a more efficient tool for text classification than language models created using neural networks [DS23].

The input of TF-IDF is a collection of tokenized documents. The tokens are typically words (*terms*), though other tokenizations can in principle also be used with TF-IDF. Let us denote the output matrix of TF-IDF by $\mathbf{W}$. Then the score for word/term t and document d is simply a multiple of two values [JM23]

$$\mathbf{W}_{t,d} = \mathrm{TF}(t,d) \cdot \mathrm{IDF}(t), \tag{2.1}$$

where:

- TF stands for *term frequency*. In the most simple interpretation, this is the number of occurrences of word t in document d. However, this makes value grows quickly with corpus size, so TF is often defined as the logarithm of the count, with 1 added to the count so that the minimal value is still $0 = \log 1$.
- IDF is the *inverse document frequency*, the ratio of documents in the corpus that contain word t (hence, IDF is independent of d). Once again, the value is often logarithmized for large corpora.

Overall, TF-IDF captures both how much a document uses a particular word, and how unique that word is in the given corpus, with both of these factors increasing the score for a particular word.

2.2.3 Generative and discriminative models

Models which classify languages can be split into two categories [JM23]:

- *generative* models, which learn to generate text from each class based on the training data. The classification is then performed by deciding which class is most likely to generate observed text according to the model. A simple example of a generative classifier is the Naïve Bayes model, which

only considers the frequencies of words w_i in each class and, given a document d, assigns to that document whichever class c has the highest posterior probability:

$$P(c \mid d) = \frac{P(c) \prod_{w_i} P(w_i \mid c)}{\prod_{w_i} P(w_i)} \tag{2.2}$$

- *discriminative* models, which only learn how to distinguish between the classes and cannot generate text. A simple example of such a model is a logistic regression classifier, see Subsection 2.3.2.

2.3 Classification

In statistical learning, all variables belong to one of the following categories [Jam+23]:

- *Quantitative* variables, which are represented by a numerical value, and can thus in principle take on an infinite number of values.
- *Qualitative* variables, which are represented by a finite set of categories.

From this distinction arise two fundamental problems of statistical learning: *classification*, i.e. predicting the value of a qualitative variable, and *regression*, where the output variable is quantitative and continuous [Bis06]. These are the emblematic problems of *supervised* learning, wherein the value of the predicted variable is known and can be used for model training, in contrast with *unsupervised* learning represented e.g. by clustering (Section 2.5).

If we can assume that each example has exactly one label, then the labels are commonly referred to as *classes* [EVW16], leading to the following:

- *Binary* classification: the class is defined by a simple binary label, marking examples as either positive or negative.
- *Multiclass* classification: there are multiple class labels.

However, there are classification problems where each example can have multiple labels, or no labels at all. In this regard, we can categorize classification problems in a different manner:

- *Single-label* classification: each example belongs to precisely one class.
- *Multi-label* classification: each example can have an arbitrary number of labels.

2.3.1 Performance metrics

Performance metrics for classification derive from the basic definitions for binary classification that can be found in many sources analyzing this topic, e.g. [G+23]. To evaluate the performance of binary classification, we compare the predictions of the classifier on a test set with the ground truth and identify the following based on the positive/negative class:

- True positives (TP) – examples that are positive and have been classified correctly,
- False negatives (FN) – examples that are positive, but have been classified as negative,
- True negatives (FN) – examples that are negative and have been classified correctly,
- False positives (FP) – examples that are negative, but have been classified as positive.

These four groups can also be arranged into a confusion matrix. We can use them to define the following commonly used metrics:

- *Accuracy* – the ratio of correctly classified examples.

$$\mathrm{ACC} = \frac{(\mathrm{TP} + \mathrm{TN})}{(\mathrm{TP} + \mathrm{TN}) + (\mathrm{FP} + \mathrm{FN})} \tag{2.3}$$

- *Precision* – the ratio of correctly classified examples among all examples that **were** classified as positive.

$$\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{2.4}$$

- *Recall* (or *sensitivity*) – the ratio of correctly classified examples among all examples that **should have been** classified as positive.

$$\mathrm{R} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2.5}$$

- *F1 score* – the harmonic mean of precision and recall.

$$\mathrm{F1} = 2 \cdot \frac{\mathrm{P} \cdot \mathrm{R}}{\mathrm{P} + \mathrm{R}} \tag{2.6}$$

Analogous metrics are also used for multiclass classification, although is not immediately obvious how to extend the above definitions to multiclass classification, since there is no single "positive" and "negative" class. The metrics can be evaluated for each class separately, treating all other classes as negatives, but this leaves open the question of combining them into a single metric representing the performance of the classifier on the entire multiclass dataset. In fact, there are several approaches to calculating performance metrics for multiclass classifiers. These can be found using the following established terminology found in many articles that report classifier performance, e.g. [DE19; YC21]:

- *Micro-averaging* – TP, FP, TN, FN are summed up across all classes and the metrics are calculated from the sums.

- *Macro-averaging* – the metrics are calculated for each class separately and for each metric, the value is calculated as the metric's arithmetic mean across all classes.
- *Weighted averaging* – similar to macro-averaging, with the exception that the means are weighted according to the prevalence of classes in the dataset.

The accuracy definition for binary classification (Eq. 2.3) is generally not used for multiclass classification, among other reasons because it results in a metric that is difficult to interpret. Instead, when considering multiclass single-label classification, we will define accuracy as

$$\mathrm{ACC} = \frac{C}{C + F} \tag{2.7}$$

where C denotes the count of correctly classified examples and F the count of incorrectly classified examples. This is a common way to define accuracy in this scenario, which is easy to interpret and analogous to binary classification accuracy.

In fact, when considering single-label classification, micro-averaging collapses the metrics defined by Eqs. 2.4 to 2.6 to a single metric corresponding to the multiclass accuracy as defined by Eq. 2.7. This is perhaps best explained by considering all ways in which an example can contribute to the confusion matrices of the classes:

- Each example is classified either correctly or incorrectly.
  - If it is classified correctly, it will be counted as a true positive once and as a true negative $n - 1$ times.
  - If it is classified incorrectly, it will be counted as a false positive once (the class it was incorrectly labeled as), as a false negative once (the class it belongs to), as as a true negative $n - 2$ times.

Hence, summing up the counts across all classes and following the notation from 2.7, we can observe that $\mathrm{TP} = C$ and $\mathrm{FN} = \mathrm{FP} = F$. Immediately, 2.4 and 2.5 become 2.7 by substituting the counts, and F1 thus becomes the geometric mean of two identical values.

2.3.2 Statistical learning algorithms

Once we acquire a vectorized representation of the corpus, either directly from a model or via a dimension reduction method, we will be interested in learning how to classify the transformed documents into pre-defined categories. While some dimension reduction methods such as LDA (Subsection 2.4.1) can also directly be used as classifiers, we may also be interested in other algorithms that are applicable in a broader range of classification problems and are known to achieve better results. On a different note, it's important to try simple approaches first. On some text datasets, even the simplest classifiers

can do almost as well as the best – see [SS23], where the difference between a Naïve Bayes model and the best tested classifier is less then 0.5 %. Another example of a simple, yet efficient classifier, which can serve as a baseline, is the K-nearest-neighbor classifier, i.e. simply predicting the most common class among K nearest neighbors in the training set. See e.g. [RBP24], where on one of the datasets, the F1 score of K nearest neighbors is within 1 % of the best tested classifier (XGBoost).

Logistic regression

Logistic regression is an example of a generalized linear model, where the output Y is related to a linear function of the input variables $X_1, X_2, \ldots, X_n$ via a link function η [Jam+23]:

$$\eta(\mathrm{E}[Y \mid X_1, \ldots, X_n]) = \beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n \tag{2.8}$$

For instance, if η is the identity function, then Eq. 2.8 describes linear regression. In case of logistic regression, the link function is the logit function

$$\eta(x) = \log\left(\frac{x}{1-x}\right). \tag{2.9}$$

Following Eq. 2.8, we must apply the inverse of η to the right-hand side of the equation to obtain the expected output. The inverse of the logit function is the sigmoid function, resulting in

$$\mathrm{E}[Y \mid X_1, \ldots, X_n] = \sigma(\beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n) = \frac{e^{\beta_0+\beta_1 X_1+\cdots+\beta_n X_n}}{1 + e^{\beta_0+\beta_1 X_1+\cdots+\beta_n X_n}} \tag{2.10}$$

The training of the model can be done by stochastic gradient descent [JM23], but commonly used implementations also use more specific solvers for non-linear problems such as LBFGS.

The above describes logistic regression in the binary classification scenario. Since the output in Eq. 2.10 is a sigmoid, its value is guaranteed to be between 0 and 1. We can thus set the output for training to 0 and 1 for negative and positive class, respectively, and set an evaluation threshold somewhere between 0 and 1 which determines the highest value which will be still classified as negative (typically 0.5, but it can also e.g. be lowered to improve the recall of the positive class).

To generalize the algorithm to a multiclass scenario, either multiple binary models are trained to detect each class, or a single model is trained using the cross-entropy loss.

Decision trees and ensembling

Ensemble models are models that consist of a set of simpler models (*weak learners*). When training an ensemble model, the training data is modified for each of the weak learners so that their predictions are slightly different. The prediction of the ensemble model is then made using the aggregate of all of

the predictions made by the weak learners. Common ensembling techniques include:

- *Bagging*, or bootstrap aggregation. Given a dataset of N samples, M new datasets are created, where each of them contains N samples. The new datasets, called *bootstrap* datasets, consist of samples which are randomly drawn from the original dataset with replacement (i.e. a bootstrap dataset can contain duplicate samples from the original dataset). A weak learner is then trained on each of the bootstrap datasets, resulting in M distinct models. These learners then act as a *committee*, where each of the models has an equal weight in the predictive decision-making – the individual predictions are averaged for a final prediction in regression, and determined by majority vote in the case of classification.

- *Boosting*. The weak learners are trained as a sequence, and instead of altering the dataset for each model, the objective function J is modified to give greater weight to examples that have been classified incorrectly by the previous learner. The learners are also not equal in the committee – their weight in the committee may decrease with the length of the sequence, or, like in the case of AdaBoost [Bis06], they may be given more weight in case their overall accuracy is better.

Ensembling works well with models that have low bias and high variance, since the variance of the ensemble model can be reduced by averaging the prediction. An example of such a model is a *decision tree* – an algorithm that works by repeatedly splitting the input space into regions until a given maximum granularity is reached. Decision trees can be used for both classification and regression [Has17]. Fig. 2.1 illustrates a simple classification tree, partitioning a 2D space into three regions for binary classification. The prediction is made by traversing the tree from the root all the way to a leaf, based on the input values of the example, and predicting whichever class is stored in that leaf. Generally, the prediction in each leaf m is made by picking the class $\mathcal{C}_i \in \mathcal{C}$ which had the highest frequency in the region during training, i.e.

$$\mathcal{C}(m) = \operatorname*{argmax}_{\mathcal{C}} \sum_{\mathbf{x}_i \in R_m} [\![y_i = \mathcal{C}_i]\!], \tag{2.11}$$

where R_m is the region represented by node m.

However, this only explains how to make a prediction in an already constructed tree, not how to build the tree itself. Building an optimal decision tree is an NP-hard problem even when the learner is able to query the learned function, let alone from random examples [KST23]. Therefore, heuristics and greedy algorithms are used to approximate the optimal tree. Commonly, a metric known as *information gain* is employed [Mit13; Ras16], which essentially measures the difference in cross entropy that can be achieved by creating a split using a particular value. An alternative to information gain is the *Gini index*.

The two following ensemble models build upon the concepts of bagging and boosting, respectively, and use decision trees as weak learners. Both of

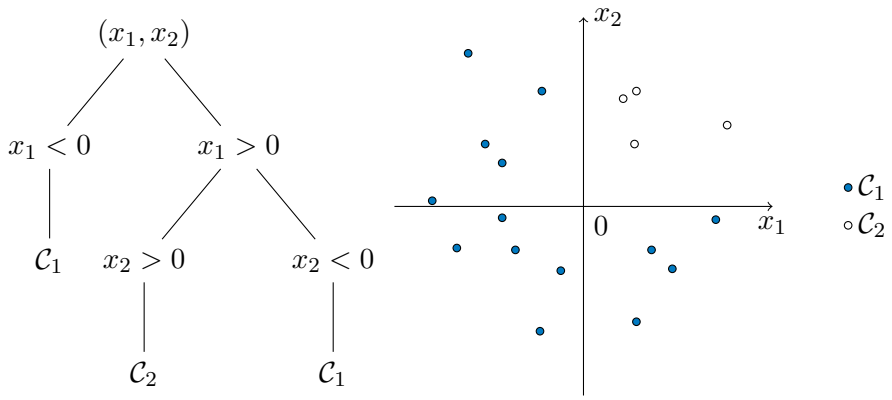


Figure 2.1: A simple classification tree.

them represent very effective approaches for approximating general functions that can rival support vector machines.

- A *random forest* is a kind of ensemble model of decision trees [Ras16], being essentially a modified version of bagging. The modification lies in a restriction on the learning of the trees – before each split, a small subset of the input variables (possibly as small as 1) is chosen at random, and the tree can only use this chosen subset for the next split. The trees are thereby forced to make suboptimal decisions, but this also results in a reduction of correlation between different trees [Has17], and ultimately the random forest performs better than a bagging model of decision trees.
- A *gradient boosting machine* is similar in principle to boosting, but uses a differentiable loss function to fit the trees onto the gradient of the loss function in each iteration, updating the predictions using a learning rate with respect to the gradient.

Support vector machine

Assuming a binary classification model which is linear in the feature space $\phi(x)$, meaning that

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b \tag{2.12}$$

The decision boundary is a hyperplane which should separate the two classes as well as possible. Since many different hyperplanes may achieve equivalent results on the training set, it may be difficult to decide which of the possible solutions has the best potential to generalize. To answer this question, the concept of the *margin* is introduced, which is the closest distance between the decision boundary and a correctly classified point [Bis06]. Given the correct labels $t_n \in \{-1, 1\}$ for all training examples n with features $\mathbf{x}_n$, maximizing the margin means optimizing the criterion

$$\arg\max_{\mathbf{w},b} \left(\frac{1}{\|x\|} \min_n [t_n(\mathbf{w}^T \phi(\mathbf{x}_n) + b)] \right). \tag{2.13}$$

Classifiers which maximize this margin are commonly called *support vector machines*, or SVMs. Since they are forced to place the decision boundary as far away as possible from the training samples, they typically achieve very good results. SVMs can also be expressed entirely in terms of dot products of $\phi(x)$, meaning that they can be trained in a transformed space defined by a *kernel function* that satisfies $K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x}) \cdot \phi'(\mathbf{x})$. Many of these kernel functions, such as the *radial basis function*, allow SVMs to also solve non-linear problems with high accuracy.

2.4 Dimension reduction methods

Dimension reduction methods are useful for compressing high-dimensional sparse matrices generated by bag-of-words representation or TF-IDF, as well as visualizing the results (typically by projecting into 2D).

Popular dimension reduction methods include:

- Principal component analysis (PCA), which finds directions in the input space which capture the most variance of the data. PCA is a general-purpose method that can be used e.g. directly on a bag-of-words representation of a document corpus.
- Latent semantic analysis, which is a method similar to PCA in that it is also based on singular value decomposition of the input, but it is a method directly adapted to NLP tasks [JM23].
- t-distributed stochastic neighbor embedding (t-SNE), which is especially suited to visualization and clustering.

While PCA is an algorithm for finding the best linear subspace, most other algorithms including t-SNE look for patterns that are only locally linear, discovering *manifolds* in the data. This section will focus on dimension reduction methods relevant to the conducted experiments, namely linear discriminant analysis, latent Dirichlet allocation and PaCMAP.

2.4.1 Linear discriminant analysis

Linear discriminant analysis, or LDA for short, is a method which, besides being useful for reducing the dimension of the independent variables, can be also used directly for classification.

LDA is derived from the assumption that each class is a random variable which follows a multivariate Gaussian distribution [Jam+23]. Let $X \in \mathbb{R}^n$ be an n-dimensional random variable, $\mu \in \mathbb{R}^n$ be the mean value of X for a given class y, and $\mathbf{\Sigma} \in \mathbb{R}^{n \times n}$ be the covariance matrix of X. Then the probability density of y at a given $x \in X$ following the multivariate Gaussian distribution is given by

$$y = \frac{1}{(2\pi)^{p/2}\sqrt{\det(\mathbf{\Sigma})}} \exp\left(-\frac{(x-\mu)^T \mathbf{\Sigma}^{-1}(x-\mu)}{2}\right). \tag{2.14}$$

Furthermore, LDA requires that the covariance matrices $\mathbf{\Sigma}$ be the same for all classes (i.e. the classes only differ in the mean values). Under this additional assumption, it can be shown that the optimal decision boundary is a linear function of X [Jam+23].

2.4.2 Latent Dirichlet allocation

Latent Dirichlet allocation is a generative probabilistic model of a corpus [BNJ03]. Given a fixed number of topics, it transforms a corpus of documents into a distribution of topics over the documents. For each document, the topics are sampled from the Dirichlet distribution and it holds that they sum up to 1 for each document, which makes them easy to interpret.

To differentiate latent Dirichlet allocation from linear discriminant analysis (LDA), the former will be referred to as LDiA, following the convention from [Hap19].

Note that LDiA assumes that the number of words in a document obeys the Poisson distribution across the corpus. This assumption will likely be false for our corpus of web page text. However, it is noted by [BNJ03] that this assumption is not important.

Latent Dirichlet allocation can be visualized as a Bayesian network (Figure 2.2) with the following hidden variables:

- α – topic distribution parameter
- θ – topic distribution for a given document, sampled from Dir(α)
- z – word distribution for a given topic and document
- β – a matrix of word distribution parameters for each topic and document

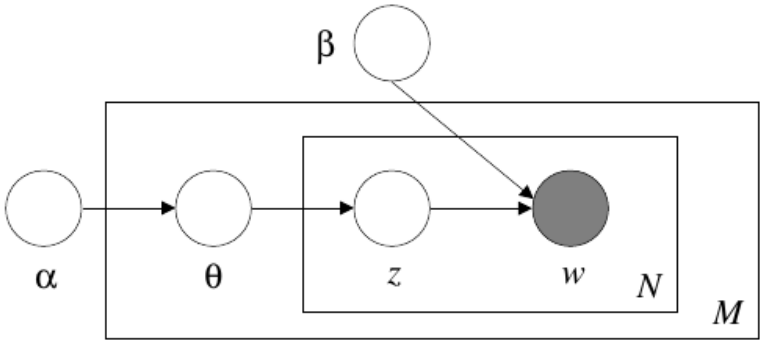


Figure 2.2: Latent Dirichlet allocation as a Bayesian network using plate notation. Source: [BNJ03].

2.4.3 UMAP and PaCMAP

UMAP (Uniform Manifold Approximation and Projection) [MHM18] is a popular dimension reduction method, created on a theoretical foundation of

topology and category theory with the aim to make the algorithm as universal as possible. UMAP is indeed suitable for both machine learning applications and visualization. Practically, the algorithm has two phases: in the first, a K-neighbor graph is constructed, i.e. a graph where each point is connected to its K nearest neighbors. Then, a layout of this graph in the projected space is computed using a force-directed layout algorithm. This basic idea of the algorithm is similar to older reduction methods such as Isomap [TSL00], but UMAP's design makes it both much faster and better at preserving the global structure of the dataset.

Although UMAP is essentially a universal dimension reduction algorithm, its output is very sensitive to changes in two hyperparameters, namely the K neighbors of the neighbor graph, and the minimum distance between two points in the projected space. Dependency on hyperparameters is generally a disadvantage in dimension reduction, since it's typically difficult to determine the correct values of the hyperparameters for a particular dataset. For this reason, several new methods have been developed to address UMAP's weaknesses. One of them is Pairwise Controlled Manifold Approximation, or PaCMAP [Wan+20]. The authors of PaCMAP analyze the various motivations for existing algorithms, note the differences between them (e.g. comparing the probabilistic origins of t-SNE to UMAP's topological derivation), and argue that none of them is essential to the dimension reduction task. Instead, they focus on explicitly capturing both local and global properties of the dataset. To this goal, PaCMAP distinguishes between 3 types of pairs of points – neighbors NB, mid-near pairs MN and further pairs FP, which are randomly sampled when initializing the algorithm. Each type of pair contributes a different summand to the overall loss function

$$\ell = w_{NB} \sum_{(i,j)\in NB} \frac{d_{ij}}{10 + d_{ij}} + w_{MN} \sum_{(i,k)\in MN} \frac{d_{ik}}{10000 + d_{ik}} + w_{FP} \sum_{(i,l)\in FP} \frac{1}{1 + d_{il}} \tag{2.15}$$

The projected space is initialized (either with PCA, or randomly) and then iteratively optimized with respect to the loss function. The run of the optimization is split into three phases, where the weights for each type of pair are different in each phase.

2.5 Clustering

Clustering is a problem in unsupervised learning where, given a dataset of N observations $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, each observation typically being a vector, the task is to discover groups of observations such that each observation is similar to other observations in the same group, and much more different from observations in all other groups. This task is also referred to as data segmentation [Has17]. Typically, the goal is to output a set partition of the dataset, so that we may unambiguously define clusters and assign each of the

observations to exactly one of the clusters (or classify it as noise). However, some algorithms which are also commonly known as clustering algorithms take a different approach, focusing on creating a hierarchy in the dataset (Subsection 2.5.2).

2.5.1 K-means clustering

The K-means algorithm is a nonprobabilistic clustering algorithm [Bis06]. It takes the expected number of clusters K as an input argument and represents these clusters with K vectors, where each of N points in the input space is assigned to the cluster represented by the vector which is the closest to that point. More formally, the algorithm performs iterations to minimize the objective function

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \mu_k\|^2 \tag{2.16}$$

where $r_{nk} = 1$ if and only if point n belongs to cluster k, otherwise $r_{nk} = 0$, and $\|\mathbf{x}_n - \mu_k\|^2$ represents the Euclidean distance between the n-th point and k-th mean vector. (r_{nk} is further constrained so that each point belongs to exactly one cluster.)

The algorithm itself consists of two steps, which alternate until convergence:

1. Update r_{nk}, so that each point n is assigned to the closest μ.
2. Set each μ_k to the centroid of all points currently assigned to cluster k.

The initial μ vectors can be chosen randomly, or r_{nk} can be initialized to assign points to random clusters and the algorithm can start with the latter step [Jam+23].

When modifying μ, it can be shown that the centroid is the optimal solution with respect to the current configuration of r_{nk}, by taking the derivative of J (Eq. 2.16) with respect to μ_k.

K-means clustering is the most popular example of a type of clustering algorithms, sometimes called "partitioning" algorithms [Est+96]. The common feature is the pre-defined set of cluster centers, which are shifted around to optimize the objective function. Another variation on this type of algorithm is K-medoids, which replaces the centroid calculation with the medoid, essentially restricting the center of the cluster to be one of the points of the dataset rather than an arbitrary point in space.

2.5.2 Hierarchical clustering

Unlike K-means and similar algorithms, hierarchical clustering has no predetermined number of clusters. Instead, it works by constructing a tree of clusters, also called a dendrogram, based on a dissimilarity measure [Has17], which typically represents the distance between the clusters.

A simple example is an algorithm that, on a dataset of N points, starts from N clusters, and repeatedly merges the closest two clusters together until

a termination condition is met [Kub17]. Such an algorithm exemplifies the agglomerative strategy of hierarchical clustering [Has17], with the opposite being the divisive strategy, i.e. starting from a single cluster and dividing it repeatedly. Let $G = \{i_1, i_2, \dots i_n\}, H = \{j_1, j_2, \dots j_m\}$ denote the two clusters whose dissimilarity is being calculated, and $d(i, j)$ the metric used to measure dissimilarity between points i and j. Once the clusters consist of multiple points, we may employ multiple approaches to calculate the dissimilarity between them:

- Single linkage, or nearest-neighbor technique:

$$d(G, H) = \min_{i \in G, j \in H} d(i, j) \tag{2.17}$$

- Complete linkage, or furthest-neighbor technique:

$$d(G, H) = \max_{i \in G, j \in H} d(i, j) \tag{2.18}$$

- Group average:

$$d(G, H) = \frac{1}{m \cdot n} \sum_{i \in G} \sum_{j \in H} d(i, j) \tag{2.19}$$

Each of these strategies can result in very different dendrograms, with single linkage and complete linkage being polar opposites, and group average representing a compromise between them.

The advantage of hierarchical clustering, when compared to K-means and similar algorithms, is that it can discern clusters of different sizes and complicated shapes. A disadvantage is that there is no clearly defined output set of clusters, and it is ultimately up to the user to figure out which clusters to extract from the dendrogram.

2.5.3 Density-based clustering

Intuitively, when looking at a scatter plot of points, what people perceive as "clusters" are usually areas with an increased density of points. To some extent, this notion can be captured by hierarchical clustering, but not in all cases, e.g. when one of the clusters has a much higher density than the other, though they are still clearly separated. To address this counter-intuitive behavior of clustering algorithms, an entire new type of algorithms arose that explicitly utilize the density of points in a neighborhood.

An early example of this type of algorithm is DBSCAN [Est+96], an algorithm which uses a minimal number of points in a neighborhood of size ϵ of a point to group the points together into dense regions, with special conditions for points on the cluster border where density is lower. These regions are called density-connected regions. The size ϵ and the minimal number of points in the neighborhood are determined by applying a heuristic onto the examined dataset, and points that don't lie within any cluster are classified as noise.

An issue with DBSCAN is that the minimal cluster density is a global parameter. Therefore, if the dataset contains a large cluster which is much more sparse than the other, much smaller clusters, DBSCAN will not be able to differentiate between them. HDBSCAN is an algorithm that amends this by combining DBSCAN's basic methodology with a hierarchical approach, noting that DBSCAN clustering can be almost exactly converted to hierarchical clustering with single linkage in the space of density-connected points [CMS13]. A special distance metric, called mutual reachability distance, is defined using the distance between 2 points and the distance to the k-th nearest neighbor for each point. A graph of mutual reachability distances between all points is computed, and a minimal spanning tree of this graph is constructed. The clusters are created by removing edges from this minimal spanning tree, proceeding in a divisive fashion to separate a cluster into two each time an edge is removed.

2.6 Deep learning

Deep learning is a subset of machine learning characterized by the following properties of used algorithms [GBC16]:

- The learning algorithm learns from relatively raw data (e.g. pixelated images or tokenized text) as opposed to learning from properties of the data obtained manually by *feature extraction* (e.g. a variable indicating whether the image contains a cat, or the sentiment expressed by a sentence). In fact, the algorithm can learn to extract these features by itself – this is known as *representation learning*.
- The algorithm can express a hierarchy of concepts, first extracting simple concepts from the raw data and then building upon that knowledge to construct more complicated concepts in the "deeper" parts.

Practically, the basic building block of deep learning is the artificial neuron, the function of which could be summarized by the following equation

$$y = g(\mathbf{w}^\intercal \mathbf{x} + b), \tag{2.20}$$

where $y \in \mathbb{R}$ is the output of the neuron, $\mathbf{x} \in \mathbb{R}^n$ is the input, $\mathbf{w} \in \mathbb{R}^n$ are the weights of the neuron, $b \in \mathbb{R}$ is the bias and g is an activation function. (The learnable parameters of the neuron are $\mathbf{w}$ and b.)

The neurons are then organized into structures called neural networks. A common example is a feedforward neural network, where the neurons are organized into layers such that the neurons in each layer only process the output of the previous layer, and their output is only passed to the next layer. This means we the weights $\mathbf{w}$ and biases b of each layer's neurons can be arranged into a matrix $\mathbf{W}$ and vector $\mathbf{b}$, and the network can be expressed as a series of matrix multiplications and element-wise applications of the activation function g. For example, a network with two layers computes the function

$$\mathbf{y} = g_2(\mathbf{W}_2 g_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b_2}). \tag{2.21}$$

This also hints at the reason why g is almost always a non-linear function in neural networks – if a linear function is used as g_1 and g_2 in Eq. 2.21, the equation simplifies to a linear equation of $\mathbf{x}$. For instance, substituting $g_1(x) = g_2(x) = x$ yields

$$\mathbf{y} = \mathbf{W_2}\mathbf{W_1}\mathbf{x} + \mathbf{W_2}\mathbf{b_1} + \mathbf{b_2}. \tag{2.22}$$

Effectively, this means that the two layers could be replaced by a single layer with $\mathbf{W} = \mathbf{W_2}\mathbf{W_1}$ and $\mathbf{b} = \mathbf{W_2}\mathbf{b_1} + \mathbf{b_2}$, the network would become a linear classifier, and there would be no point in creating the "layered" structure, which is very important for achieving the goals of deep learning as outlined at the start of this section.

The networks are then commonly trained by methods similar to gradient descent, using the derivative chain rule to calculate the learning step for all learnable parameters. A lot of the calculations during the evaluation of the chain rule for each learnable parameter are repetitive. For this reason, the chain rule expressions are essentially organized into a graph, allowing us to re-use previously obtained results in a fashion similar to dynamic programming. This procedure is known as *backpropagation* and the cached results are known as *backward messages* (contrasting with *forward messages* obtained when making predictions).

To calculate the gradient, we need a differentiable loss function that can quantify how wrong the predictions are. A commonly used loss function for classification is the *cross-entropy loss* [JM23]

$$\ell = -\sum_{k=1}^{K} \mathbf{t}_k \log(\mathbf{y}_k), \tag{2.23}$$

where $\mathbf{t}$ is a one-hot vector indicating the true class label. However, a further problem arises from the fact that a neural network can output any vector of real numbers. This makes the output unstable and difficult to interpret as a prediction of a class. To fix this, we attach a *softmax layer* to the end of the network, using $\mathbf{y}$ to compute $\mathbf{p}$:

$$\mathbf{p}_k = \frac{\exp(\mathbf{y}_k)}{\sum_{c=1}^{K} \exp(\mathbf{y}_c)} \tag{2.24}$$

This guarantees that the new output vector $\mathbf{p}$ sums up to 1 and $\mathbf{p}_k \in [0, 1]$ for all k, making the output easy to interpret as a vector of K probabilities that indicate the network's belief that the example belongs to class k. Furthermore, using a softmax layer with cross-entropy loss makes it possible to simplify the backward message to a simple element-wise difference between $\mathbf{p}$ and $\mathbf{t}$, making this combination also relatively quick to train.

Feedforward networks are very powerful, but their training quickly becomes computationally infeasible as the size of the layers is increased and more layers are added. For this reason, more deliberate network architectures have been created that restrict the network's complexity and allow it to focus on aspects of the data that we want to capture. It's also crucial to consider how

the data is actually passed to the network – in other words, how we preprocess and chunk the data before feeding it to the network as input. This is why the rest of this section will focus on network architectures and applications relevant to natural language processing.

2.6.1 Word2Vec and Doc2Vec

Word2Vec is an algorithm that can be used for creating a simple language model from a corpus, as well as word *embeddings* – vectors that encode the word in a Euclidean space with a pre-determined number of dimensions n. Famously, Word2Vec embeddings can have properties that align with the language's semantics – not only are words similar in meaning close to each other, but directions in the space encode specific changes in meaning and relationships between the words, e.g. the difference between the vectors of "Paris" and "France" is very similar to the difference between "Italy" and "Rome" [Mik+13].

It could be argued that Word2Vec is not strictly a *deep* learning algorithm, because it doesn't use a layered network structure. Nonetheless, it was included here because its approach to learning is quite similar – representing data using vectors, and using gradient descent instead of trying to solve directly or using more complicated optimization algorithms.

Word2Vec has been most successful with the *skip-gram* learning technique, where the algorithm is trained on fixed-length sequences of words from training data. The aim is to predict the context for a given word, i.e. other words that are likely to precede and/or follow it in text. This means that it also creates a model of meaning, since words that are similar will most likely be seen in a similar context, and thus Word2Vec creates the "semantic" structure in the embeddings.

Internally, Word2Vec keeps two n vectors for each word w – an input vector $\mathbf{v}$ and an output vector $\mathbf{v}'$ (also known as the context vector [JM23]). Given an input word w_I, the predicted function is the probability that a word w_O from the vocabulary W occurs in its context. Naïvely, this probability would be computed as [LM14]

$$p(w_O \mid w_I) = \frac{\exp({\mathbf{v}'_{w_O}}^\intercal \mathbf{v}_{w_I})}{\sum_{w \in W} \exp({\mathbf{v}'_w}^\intercal \mathbf{v}_{w_I})}, \tag{2.25}$$

i.e. the softmax of the dot products of the input word's input vector $\mathbf{v}_{w_I}$ with the output vectors $\mathbf{v}'$ of all words. Since the vocabulary W is typically quite large, this softmax is too expensive to be practical, especially for training. However, since the denominator in the softmax does not depend on w_O, it can be omitted when maximizing or minimizing $p(w_O \mid w_I)$. Furthermore, we can replace the exponential function with a different monotonously increasing function that prevents the risk of numerical overflow – the sigmoid function σ is often used in practice for this purpose. In other words,

$$\max_{w_O} p(w_O \mid w_I) = \max_{w_O} \exp({\mathbf{v}'_{w_O}}^\intercal \mathbf{v}_{w_I}) = \max_{w_O} \sigma({\mathbf{v}'_{w_O}}^\intercal \mathbf{v}_{w_I}) \tag{2.26}$$

To train the skip-gram model, *negative sampling* is commonly used. In each training step, besides maximizing the probability of a word w_O in the current context, K words are selected from the other parts of the corpus at random and the probability of these randomly drawn negative examples is minimized at the same time. This leads to minimizing the loss function

$$\ell = \log \sigma(-\mathbf{v}'_{w_O}{}^{\intercal}\mathbf{v}_{w_I}) + \sum_{i=1}^{K} \log \sigma(\mathbf{v}'_{w_i}{}^{\intercal}\mathbf{v}_{w_I}), \tag{2.27}$$

since σ is an odd function and the probability itself can replaced by its logarithm, as is common practice in maximum likelihood estimation. Negative sampling is visualized in Fig. 2.3.

Although Word2Vec is a fairly cheap and effective algorithm for language modeling, it has several key disadvantages, among others the following:

- The algorithm understands what words are typically *close to* each other, but it doesn't understand the role that their *position* plays in their meaning. To be more precise, the algorithm doesn't use any positional encoding, at least in its commonly used variants. Therefore, the contexts "Gavrilo Princip *killed* Franz Ferdinand" and "Franz Ferdinand *killed* Gavrilo Princip" mean the same thing to the algorithm – it might be able to distinguish who is the victim based on other contexts, but both of these sentences will nonetheless appear equally valid when encountered.

- Word2Vec struggles with words that can have multiple meanings, since it assumes that each word can be represented with a single vector. For example, the vector for the word "lead" will ultimately somehow have to encode the meaning of a heavy metal, a leash, a hint, as well as leadership.

Both of these are explicitly addressed by more complex architectures, such as transformers (subsection 2.6.3), which use position encoding to respect the order of the words in the context, and can use the attention mechanism to infer the meaning of an ambiguous word from the context.

Word2Vec can also be extended to not only produce embeddings of words, but also embeddings of sentences, paragraphs or documents. Two modifications of Word2Vec for encoding longer segments of text, proposed by the original authors of Word2Vec and originally called *Paragraph Vector* [LM14], are now commonly known as *Doc2Vec*. In both of these variations, the documents (paragraphs) are treated almost exactly the same as words, being represented by a vector pair (though the dimension of the document space does not have to be the same as the word embedding dimension), and used as input for the softmax classifier. The difference is in how the words are used together with the documents:

- In the *distributed memory* model, the classifier receives context words from the paragraph together with the paragraph, and predicts a word in the same context.

- In the *distributed bag of words* model, the classifier's only input is the paragraph vector. The algorithm is then tasked to predict words from contexts that are contained within the paragraph. This model is more similar to the skip-gram model – the only difference is that the input word vector w_I is replaced by the paragraph vector.

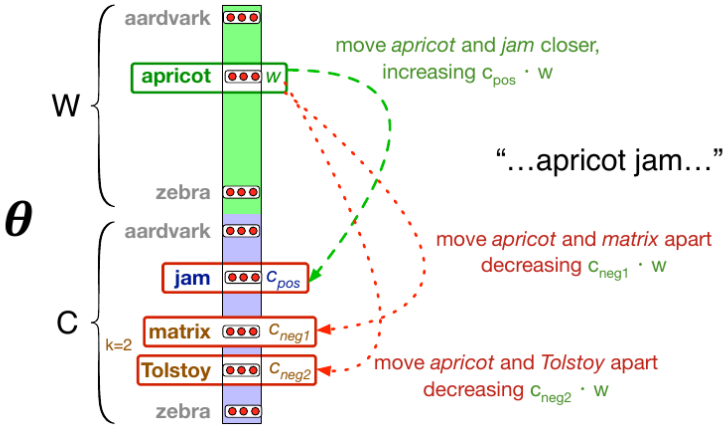


Figure 2.3: A visualization of Word2Vec skip-gram learning from [JM23].

2.6.2 LSTM

The downside of approaches such as the skip-gram model is that they can only capture a sequence of limited length, which is typically quite short. This means that they are not able to natively model relationships that are expressed by longer sequences of text than their context window. It's possible to help the skip-gram model in this regard by prioritizing negative samples that are further away in the text from the current context [Mik+13], but this doesn't fix the core issue.

A network architecture for sequences of arbitrary length, which is common in many fields, is the *recurrent neural network*, or RNN. RNNs store a vector commonly denoted by $\mathbf{h}$ from the last computation performed by the network. Each next step of the sequence with index t is then computed using the saved state from the previous step, $t-1$ [JM23]:

$$\begin{aligned} \mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \\ \mathbf{y}_t &= f(\mathbf{V}\mathbf{h}_t) \end{aligned} \tag{2.28}$$

where $\mathbf{x}_t$ and $\mathbf{y}_t$ are the input and output of the network at step t, g, f are the activation functions, and $\mathbf{U}, \mathbf{V}, \mathbf{W}$ are matrices of weights. This means that $\mathbf{h}_t$ depends on $\mathbf{h}_{t-1}$, which in turn depends on $\mathbf{h}_{t-2}$ and so on, and thus the network can save information from any input in the sequence and keep it no matter how long the sequence is.

However, experiments have shown that while RNNs can in theory hold information for an arbitrary number of steps, and will do so when explicitly

trained for a task that requires it, they tend to forget the information very quickly, only keeping it for a couple of steps [JM23]. LSTM (*long short-term memory*) is a network architecture that addresses the problems with RNNs by adding an extra context vector $\mathbf{c}_t$, which is also "remembered" throughout the sequence just like the hidden state $\mathbf{h}_t$. However, unlike in the case of $\mathbf{h}_t$, the network contains an explicit mechanism of *gates* which allows it to control what is saved into $\mathbf{c}_t$, and when the information can be forgotten.

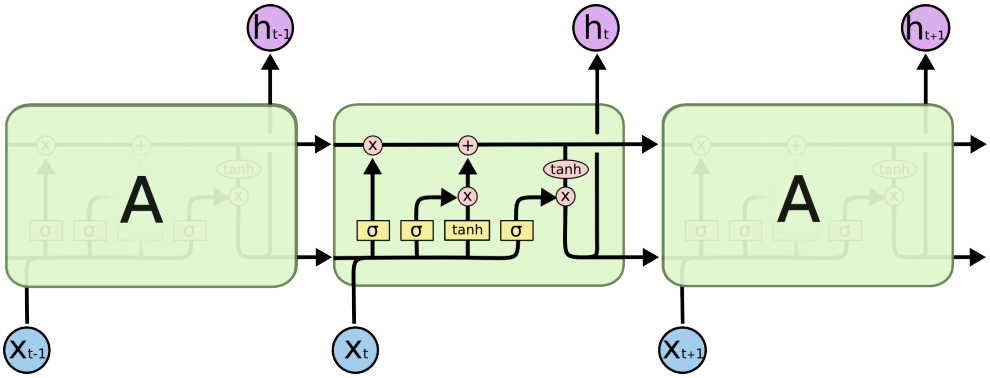


Figure 2.4: A schematic showing the flow of information through an LSTM network and the gates mechanism. The upper stream passed along the sequence from left to right is the context vector $\mathbf{c}$, the lower is the state $\mathbf{h}$. The symbols with an orange background in the image depict a layer of neurons followed by the symbolized function, whereas the ones with red background convey a simple application of the function with no trainable parameters. The "x" symbol stands for element-wise multiplication. Source: [Ola15].

The LSTM contains three of these gates, all of which have the following features in common:

- The gate takes both the learned state of the sequence $\mathbf{h}_{t-1}$ and the current element of the sequence $\mathbf{x}_t$ as input. The two inputs are then multiplied by the gate's weights and added together or concatenated, depending on the particular implementation. After that, the sigmoid function σ is applied element-wise to the result, meaning that each element in the gate output is a number between 0 and 1.
- The gate's output then acts as a *mask* or filter, i.e. it is multiplied element-wise with a different vector, allowing the network to de-emphasize certain values via the sigmoid outputs.

The gates, which are visualized in Fig. 2.4, are frequently named after their function in the network:

- *Forget gate*: This is the leftmost gate in Fig. 2.4. It is the first gate to access $\mathbf{c}_{t-1}$, removing context that is no longer needed when receiving $\mathbf{x}_t$ in state $\mathbf{h}_{t-1}$ and forwarding the relevant context to the add gate.
- *Add gate*: This is the gate in the middle of the block in 2.4. $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ are first run through an "RNN" embedded in the LSTM unit, using the

hyperbolic tangent activation function. Then, the output of this block is filtered by the add gate to sift out all that is not new context, and added to the output of the forget gate.

- *Output gate*: This is the final gate in the right section of the block. The output is also used as the state passed to the next step, $\mathbf{h}_t$. It is created by first applying the hyperbolic tangent function to the output of the add gate, and then filtering it using the output gate, which has the same inputs as the previous gates, differing only in the learned weights.

Note that the architecture is not standardized and the network can also be trained successfully with some variations to the structure described above. See e.g. [Gra13] for a description of an LSTM network, where all of the gates also have access to $\mathbf{c}_{t-1}$ besides $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$.

LSTMs are also viable in an *encoder-decoder* setting [Hap19], which is ubiquitous in language translation, and many other situations where we want to convert one representation of an example into another. The idea is to feed the entire input sequence through an LSTM encoder, and use the final state $\mathbf{h}$ to generate the output sequence with a separate LSTM decoder trained specifically for this task.

2.6.3 Transformer models

Transformer models probably need no introduction nowadays, being the foundation of almost any large model known to the general public. The key component of the transformer is the *attention* mechanism, which actually predates transformers, having been previously used in sequential models similar to the LSTM network. The transformer removes the sequential processing and replaces it with a context window with fixed length, reflecting the original order of the sequence only by applying positional encoding to the inputs, and using exclusively attention and feedforward networks as building blocks for its architecture. Despite losing the ability to capture relationships at arbitrary distance by discarding the recursive part of the networks, the transformer has been found to be a well performing model since the early experiments [Vas+17]. Importantly, the lack of explicitly sequential processing also has a benefit in making all computations in the transformer highly parallelizable.

When a sequence enters the transformer, it is first embedded in space in a similar fashion to a Word2Vec embedding. Then, positional encoding is applied to these embeddings, which can be done using sinusoid functions. This allows the transformer to be aware of the sequential nature of the input text, without forcing the computation itself to be sequential. After the position encoding, the embeddings are then processed by a sequence of repeated attention blocks and feed-forward blocks. Finally, softmax is applied to the output, so that the output can be interpreted as a vector of token probabilities.

The attention mechanism itself is usually conceptualized as a set of attention *heads*, which operate in parallel. Each head has a *key* matrix $\mathbf{K}$ and *query*

matrix $\mathbf{Q}$, which have the same dimension and store information about the relevance of the query to the contextual meaning of the key token. These two matrices are multiplied together to create a kind of "score" that signifies the influence of the key on the meaning of the query. The scores are then normalized and multiplied by the *value* matrix $\mathbf{V}$ to determine how the embeddings should be updated in this context. This is described by the famous equation from the original transformer paper

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \tag{2.29}$$

This equation describes a single *self-attention* head. Crucially, attention can be used both as self-attention, where the output format is the same, or *cross-attention*, whereby attention can transform the data into an entirely different structure [Dev+18]. The original transformer architecture was an encoder-decoder model, but the attention mechanism is now also used in encoder-only models like the BERT family, and decoder-only models like GPT [TWW22].

Chapter 3

Overview of existing tools

It should be noted that the grouping of the listed tools by utilization is somewhat arbitrary, as some of them have a broader scope and can be used for several of the tasks listed in this chapter. If a tool can be used for multiple tasks, it can be found under the task for which it was deemed most likely to be useful.

This chapter largely describes non-scientific projects which cannot be cited very meaningfully, so in those cases, URLs to the relevant webpages are provided in footnotes instead. The validity of the links was last checked on May 21, 2024.

3.1 Databases

There's plenty of options available when it comes to storing a knowledge base and making it available for querying. In an extreme case, even a relational database could be used, e.g. **PostgreSQL** offers support for JSON[1], which could be used for representing diverse entities without the need to complicate the database structure in the process. Similarly, a native JSON database like MongoDB could be used for this purpose – **MongoDB** in particular even offers support for querying graphs embedded in its collections[2].

Nonetheless, there are also frameworks that explicitly target graph structures and/or knowledge engineering. **Neo4j**[3] is a popular and versatile graph database. It can be updated and queried using Cypher, which was described in 2.1.2, and it also has a Java interface that allow programmatic access to the database. Databases that are built for RDF data include Apache **Jena**[4] and Ontotext **GraphDB**[5].

[1] https://www.postgresql.org/docs/current/datatype-json.html
[2] https://www.mongodb.com/resources/basics/databases/mongodb-graph-database
[3] https://neo4j.com/
[4] https://jena.apache.org/
[5] https://graphdb.ontotext.com/

3.2 Web scraping

All listed packages and libraries are implemented in Python and intended for use in Python, unless specified otherwise.

The **requests** library is a commonly used tool for interacting with websites. It enables simple execution of most HTTP methods and easy addition of headers and cookies to the performed requests. However, it is somewhat impractical for web crawling, because it doesn't offer supports for asynchronous requests. This has been amended by **grequests**[6], a package that enables sending requests in batches of a defined size via gevent and allows interaction with the `Request`/`Response` objects which is analogous to `requests`.

Scrapy[7] is a more advanced framework for scraping and crawling. The algorithm for crawling and extracting the results is defined by the methods of a `Spider` object in an automatically generated project directory, while the framework itself handles the priority queue of the requests and the saving of the results. The behavior of the crawl can be altered by modifying the `settings.py` file of a given project. The output of a crawl may be saved into common formats such as CSV or JSON without the need to directly modify the source code (the format is specified when launching the crawl).

The Scrapy developers also provide various plugins that extend the basic framework, most notably **Playwright**[8] support.

Minet[9] is a specialized web data mining library.

Crawlee[10] is a scraping and browser automation framework for JavaScript, built on top of Node.js. It offers similar functionality to Scrapy, providing an API for building crawlers. It facilitates easy switching between pure HTML-based crawling and the use of a browser and rendering – this is a native feature of the framework, making it more flexible for browser-based scraping than Scrapy.

3.3 Parsing HTML

Many packages for web crawling and/or web content extraction already have a HTML parser included, so it's likely that a separate HTML parsing library will not be required. HTML parsers convert the raw HTML into a DOM[11] representation, which can then be iterated over or filtered. They frequently employ standardized selection methods such as CSS selectors or XPath to allow the user to select elements from the website and iterate over them. The most popular standalone library for parsing HTML is **BeautifulSoup**[12], which understands CSS selectors and is compatible with several parsers such

[6] https://github.com/spyoungtech/grequests
[7] https://docs.scrapy.org/en/latest/
[8] https://github.com/scrapy-plugins/scrapy-playwright
[9] https://github.com/medialab/minet
[10] https://crawlee.dev/
[11] *Document Object Model*, representing the HTML document as a tree of elements.
[12] https://www.crummy.com/software/BeautifulSoup/bs4/doc/

as **lxml**. **Parsel**[13] is a HTML, XML and JSON parser developed as a part of Scrapy, that can also be used as a standalone library. Compared to BeautifulSoup it has more support for using standardized selectors defined by the CSS and XPath standards.

3.4 Rendering and browser emulation

Automated browsers are a widely used tool, common e.g. in web application testing. It is common for modern websites to be generated at least in part dynamically, with the browser executing embedded scripts that fetch data and render more content than is present in the raw HTML. For this reason, automating browsers can be beneficial in web scraping as well, possibly retrieving more content and making the crawler less likely to be detected as a bot.

Traditionally, the most commonly used browser emulator is **Selenium**[14], which can be used via APIs for many common languages including Python and Java. However, various alternatives to Selenium are gaining traction, which are built in JavaScript but also available for Python. The most notable examples are **Playwright**[15], an open-source engine developed by Microsoft, and **Puppeteer**[16].

A downside of using a browser emulator is that rendering pages while scraping, instead of extracting content from raw HTML using tools described in Section 3.2, creates significant overhead. For this reason, it might be beneficial to combine Selenium with other tools and only use it when necessary. A successful implementation of a crawler integrating Selenium with Scrapy was reported by [Wu+20].

3.5 Content extraction

To successfully extract knowledge from a website, it will be important to accurately detect relevant text contained in its pages. Alternatively, we may state this task as discarding sections of the websites which recur frequently on a given website, or are not related to the main content of the page (e.g. advertisement). In literature, such redundant content is often referred to as *boilerplate* – hence, content extraction is equivalent to boilerplate removal [VGE18].

Following [ZW21], we may split content extraction tools into three groups:

- Rule-based methods, which use various heuristics to predict whether a tag is boilerplate, such as stopwords frequency and detecting clusters of more intensive text tags in the DOM tree. They are both general and fast, but sometimes not very accurate.

[13] https://pypi.org/project/parsel/
[14] https://www.selenium.dev/
[15] https://playwright.dev/
[16] https://pptr.dev/

- Website-based methods, which are more specifically tuned to particular targets.
- Machine-learning methods, which use more complex approaches, see [ZW21] or [LAK20].

The challenge in extracting text from website is to filter out text that is largely irrelevant to the website's actual content, like advertisements and terms of service. Currently available open-source tools and libraries typically use heuristics to discern text from boilerplate content, but are unaware of the semantics of the text.

Compared to the topics which have been discussed earlier in Chapter 2, the existing research is rather difficult to make sense of – the articles unfortunately don't provide their actual implementation of the algorithms to the public and the commonly utilized and tested libraries are completely unrelated to them. Examples of articles that claim to use more advanced content extraction methods include: [UL18], where position of an element on the screen is encoded into a hash code and then the elements are processed using the hashcode and empirical formulas; and [Gan23], where a combination of K-nearest neighbors and a genetic algorithm is utilized.

The package **boilerpy3**[17] provides an easy-to-use interface for content extraction. It includes several `Extractor` objects tuned for different tasks. It is a relatively simple library without advanced NLP functions, nonetheless, it can compete with the other listed libraries for the task of article extraction, as reported by the authors of Trafilatura. `boilerpy3` originated as a Python port of the Java library Boilerpipe, which is also still available.

jusText[18] is an example of a more complex package specializing in content extraction. It has rich support for detecting many different natural languages and its authors provide a web frontend where one may visualize its output for a given webpage and target natural language. jusText has also been ported to other languages such as C++ and Java.

Trafilatura[19] is a complex package for acquiring textual information from the web, encompassing crawling and scraping, content extraction and even text embedding for an advanced analysis.

All of the above methods assume that we want to extract text content and use machine learning methods to extract knowledge from that text. However, there are also tools that web developers can use to embed structured data into the HTML source code. This data is not shown to the end user – instead, it is provided to transfer information to crawlers and other scripts that visit the webpage. **Schema.org**[20] and **Open Graph**[21] are the most common protocols used for this purpose. *RDFa* and *Microdata* are extensions to HTML syntax that allow this embedding of structured data.

[17] https://github.com/jmriebold/BoilerPy3
[18] https://github.com/miso-belica/jusText
[19] https://trafilatura.readthedocs.io/en/latest/
[20] https://schema.org/
[21] https://ogp.me/

3.6 Topic detection

The **scikit-learn**[22] library contains implementations of many algorithms mentioned in sections 2.3 and 2.4. Many of these implementations are directly intended for text analysis and can be used with other freely available Python libraries like **nltk**[23].

Several Python packages combine multiple tools, such as the pretrained models above and clustering algorithms, and aim to provide a unified pipeline for topic modeling from text corpora. These include **BERTopic**[24] and Top2Vec, which is described below in Subsection 3.6.1.

The Universal Sentence Encoder (**USE**) [Cer+18] is a family of pretrained models, using deeper architectures to create embeddings from sentences of text. It also encompasses multilingual models [Yan+19], which can assess similarity of text in different languages and supports 16 common natural languages, including Spanish, Chinese or Arabic.

There are many popular pretrained models based on the BERT architecture, like RoBERTa [Liu+19]. In topic modeling, a popular BERT variant is **SBERT** [RG19], which is available in both Top2Vec and BERTopic.

GloVe[25] and **FastText**[26] are word embedding methods that can be used instead of the Doc2Vec model described in Subsection 2.6.1.

3.6.1 Top2Vec

Top2Vec is a Python package integrating the entire process of extracting topics from a corpus of text, as well as a composite algorithm for discovering topics from raw documents [Ang20]. It is probably best understood as a sequence of the following three operations:

1. The documents are tokenized, and all documents and words are embedded into a shared vector space. This means either that a model is trained/finetuned on the corpus, or that the embeddings are simply retrieved from an existing model.

2. Dimension reduction is applied to the embedding space. This is essentially a preprocessing step, making topic modeling easier than in the original, high-dimensional space.

3. In the reduced space, clustering is performed on the document vectors. The document clusters are then interpreted as topics, with the centroid of a particular cluster defined as the topic vector. Words that are the closest to the topic vector are taken to best represent the themes inside this topic. For each document, a topic distribution can then be calculated

[22] https://scikit-learn.org/stable/
[23] https://www.nltk.org/
[24] https://maartengr.github.io/BERTopic/index.html
[25] https://nlp.stanford.edu/projects/glove/
[26] https://fasttext.cc/

based on the document's closeness to the topic vectors, yielding a more nuanced view of the document's content than the initial cluster hierarchy.

In the first phase, the Top2Vec package supports either training a Doc2Vec model, or using a pre-trained model. The Doc2Vec implementation is imported from Gensim [ŘS10], and the continuous bag of words variant of the algorithm is used (see Subsection 2.6.1). Since this variant of Doc2Vec does not train word vectors, which are needed for the interpretability of Top2Vec result, the word vectors are trained separately on the same data, which is enabled by the Gensim implementation. Gensim also implements the distributed memory model of Doc2Vec, but this can only be used by editing the Top2Vec source code. As for pre-trained models, the package offers easy access to some models available through TensorFlow, namely Google's Universal Sentence Encoder and SBERT. A custom pre-trained model can also be used, but it must be ensured that it accepts the documents as a simple list of strings, returning an iterable of vector embeddings for each of the input documents. For example, Gensim's Doc2Vec has an `infer_vector` function that allows it to embed a document it has not been trained on, but it assumes that the documents passed to it are already tokenized. This means that in order to use `infer_vector`, the documents first have to be preprocessed:

```
from top2vec_modified.Top2Vec import (
    Top2Vec,
    default_tokenizer as tokenize
)
from gensim.models.doc2vec import Doc2Vec, TaggedDocument

def embed_batch_doc2vec(docs: list[str], doc2vec: Doc2Vec):
    return [doc2vec.infer_vector(tokenize(i)) for i in docs]

doc2vec = Doc2Vec.load("doc2vec_test.model")

documents = ["doc 1", "doc 2"]

top2vec = Top2Vec(
    docs=documents,
    embedding_model=embed_batch_doc2vec
)
```

In the second phase, Top2Vec uses UMAP (see Subsection 2.4.3). The package does not support the use of other dimension reduction methods. Likewise, clustering is also done using a single algorithm – HDBSCAN, which is briefly described in Subsection 2.5.3.

3.7 LLMs and AI search engines

Several companies are currently developing and operating experimental search engines which use large language models to analyze and summarize the retrieved results and answer questions in natural language based on the results. However, in this case, the details of the implementation (e.g. how the results are indexed and preprocessed, how the LLM retrieves the results from the database of the operating company) are confidential and not much is disclosed about the inner workings of the engine. Therefore, we cannot directly draw inspiration from the methods used by these engines. Instead, they can be used as a baseline for assessing the efficiency of classifying the businesses and other tasks.

Examples include **Perplexity.ai**[27] and **Phind**[28]. LLM services are also almost always available via an API.

Various online services also offer a range of different models, including state-of-the-art large models, allowing for comparisons between them. These include the **LMSYS Chatbot Arena**[29] and **HuggingChat**[30].

Large models whose weights are openly available can also be run locally. Large communities exist which are dedicated to running open-source LLMs and building tools that facilitate the deployment of LLMs by non-professionals. The **Ollama** project[31] has a wide range of available models and is relatively user-friendly.

3.8 Workflow management systems

When creating a pipeline or ensemble consisting of several models from different sources, there will inevitably be many compatibility issues due to different expected input/output formatting or due to the utilization of entirely different frameworks or even programming languages by the constituent models. Some tasks, such as tracking loss during training or visualizing the results of the model, are predictably going to be a necessity in almost all models, and they should therefore be made as easy and convenient as possible. What's more, the deployment of the resulting system is in practice also frequently tasked to the same team that designs and trains the model – a 2021 survey found that 40% of ML professionals are simultaneously responsible for both models and infrastructure [Mak+21].

What arises from this is a set of tasks that are not strictly related to machine learning proper, but are nonetheless a necessary component of the development process, focusing on reliability, scalability, and managing the lifecycle of the product. This area is often called MLOps, an analogy to the DevOps of software development [BHN23; MG22].

[27] https://www.perplexity.ai/
[28] https://www.phind.com/search?home=true
[29] https://chat.lmsys.org/
[30] https://huggingface.co/chat/
[31] https://ollama.com/

The most common software in this domain is **MLflow**[32], alternatives for building pipelines include Spotify's **Luigi**[33] or Apache **Airflow**[34]. **Gerapy**[35] is a framework for managing crawlers written using Scrapy.

[32] https://mlflow.org/
[33] https://github.com/spotify/luigi
[34] https://airflow.apache.org/
[35] https://github.com/Gerapy/Gerapy

Chapter 4

Methodology

In this chapter, we will define the problem we aim to solve in our experiments and propose several approaches towards achieving that goal.

4.1 Problem statement

In general the inputs and the expected outputs of the experiments are the following:

- Input: a set of merchants (businesses), where each merchant has an assigned URL that should represent it to the user, and a target category.
- Output: a prediction of the merchant's category.

The URLs in the input data are selected by human workers. It is preferable that the URL of the business lead directly to the website of the merchant. However, in case the merchant doesn't operate its own website, the URL may also lead to the entry of the merchant on a website that aggregates information about businesses, or to the merchant's profile on a social media site.

The input category labels are also determined by human specialists. Whenever possible, we will use the manually created labels as the ground truth for our experiments. This means that we will be dealing mostly with supervised learning and classification.

Additionally, there is a hierarchy of tags that are associated with each category. For instance, once we know that a merchant belongs to the category *Food And Drink*, we may further classify it with tags *Restaurant*, *Cafe*, and so forth. If we choose the tag *Restaurant*, we can once again add tags that are the children of this tag, like *Czech Restaurant*, *Asian Cuisine* and so on. The hierarchy is visualized in Appendix B and will be formally described as part of the knowledge base proposal. Using this entire tag hierarchy would amount to a more complex multilabel classification problem that could be decomposed in a recursive fashion as indicated by the example. However, in order to solve the deeper levels of the hierarchy, we need to first learn to distinguish the top-level categories, so this is what we will prioritize.

Given this information, the expected procedure of the experiment is as follows:

1. Crawl the websites of the businesses provided in the URL links and collect the contained information.
2. Process the content of the websites to extract relevant information/features.
3. Train the model on this content, either directly utilizing the provided labels, or using an unsupervised approach.
4. Make the predictions using the model.

Alternatively, we may use an external source of knowledge to categorize the business without processing the content of the website. Both of these approaches will be elaborated upon in this chapter.

Additionally, due to the focus of this thesis, we will propose a knowledge base for storing the extracted predictions, demonstrate the creation of a knowledge base and insertion of the predictions.

4.2 Main pipeline

As already foreshadowed by the theoretical introduction, we expect to mainly use the text content of the websites and natural language processing methods. In particular, it appears that a valid approach might be to extract sequence embeddings from a language model and then train a classifier on the embeddings.

Even though the website content might be retrievable from some publicly available datasets such as Common Crawl, we prefer to do the crawling ourselves. This is because for our use case, the data should be as up-to-date as possible, and some small business websites might not be contained even within large crawls.

The data flow in the proposed pipeline is visualized in Figure 4.1. Note that there are two events that may make the pipeline unable to output a category prediction for a given business: either the website is unreachable, or all of the website's content is rejected by the text extraction algorithm (it may only contain images, or a text in an unaccepted language). Furthermore, an important feature of the pipeline is that the categories (i.e. the labels of the input URLs) only enter the pipeline at the very end when training the classifier, and all steps until that point can be considered unsupervised learning. In other words, the processing of the extracted document corpus is almost identical to Top2Vec (see also subsection 3.6.1), with the exception that Top2Vec is unsupervised all the way to the end – it learns the categories (topics) by applying clustering to the embeddings of the documents, rather than by recognizing patterns in the embeddings using provided labels. Lastly, much like in Top2Vec, "model training" can in principle be replaced with getting the document embeddings from a pretrained model, though this will obviously have an impact on the quality of the embeddings.

The prediction will be made entirely using data acquired using web scraping and the known URLs for each merchant, without using other information like MCC codes which is also available for the payment data.

The steps of the pipeline could be described in more detail as enumerated below:

1. For each merchant, perform a shallow crawl (see Section 4.3) of the merchant's domain. Save all reached webpages to local storage.
   - Exclude all merchants whose domains were not reached by the crawler.
2. Extract the text from all saved HTML pages and perform boilerplate removal using a text extractor. Output a single document of non-boilerplate text for each domain.
   - Exclude all merchants that have no text left after the boilerplate removal.
3. Create a high-dimensional embedding of the documents corresponding to the merchants' domains.
4. Apply a dimension reduction (manifold learning) algorithm on the set of embeddings to create a less sparse embedding.
5. Split embedded documents into training and test set. Train a classifier on the training set and evaluate performance on the test set.

The first and second step require flagging a part of the input data as undecidable and removing it from the training data. It's likely not possible to avoid excluding some of the input data due to various random factors – e.g. the domain name may become invalid or there may also be errors in the input data itself.

The scale of the input data will require dealing with hundreds of thousands of input URLs, which can easily result in a number of saved HTMLs pages in the order of magnitude of millions or more. Even when restricted to smaller datasets, managing the saved HTML files may be surprisingly difficult. Several aspects of storing the data should be considered:

- **Compression**: HTML can be compressed quite well, common formats such as gzip and zip should suffice to reduce the memory consumption of the stored data, while not slowing down access significantly.
- **Organization**: It is inconvenient to deal with folders that contain hundreds of thousands of items, so some degree of hierarchy is necessary.
- **Referencing**: URLs contain characters that cannot be used in the name of a file, so a different way to refer to a locally saved HTML page must be defined.

4.3 Scraping algorithm

The motivation for this naïve scraping procedure is that most business websites contain a page describing the history and nature of the business (usually titled "About Us" or similar). This page is usually the most useful for determining the business's category, and is typically reachable from the root page of the domain. The text contained in such a page is especially crucial if the business doesn't offer its products/services online, and it is also of particular interest since the utilized text extractors are biased towards longer text that contains entire sentences.

For each URL, the algorithm simply accesses the root of the URL's domain, saves the pages, then retrieves all pages in that domain that are reachable from from the root pages and saves them, too. The parsing procedure is summarized in the provided pseudocode, where:

- `EXTRACT_DOMAIN_ROOT` is a string function (regular expression) which converts the URL into the domain name (e.g. `https://www.tesco.com/special-offers/` into `tesco.com`),
- `SAVE_FILE` simply stores the raw HTML locally, though there are some caveats such as that the URL usually cannot be used as a file name – see Subsection 5.1.1 for a more detailed discussion.

```
Input: list of urls U
Output: locally stored files containing raw HTML
forall url ∈ U do
    domain ← EXTRACT_DOMAIN_ROOT(url);
    html ← HTTP_GET(domain);
    SAVE_FILE(html);
    forall link ∈ CSS_SELECT(html, "a::attr(href)") do
        if link ∈ domain then
            link_html ← HTTP_GET(link);
            SAVE_FILE(link_html);
        end
    end
end
```

The scraping algorithm has some issues, such as that the root page of the domain can in principle contain any number of links. This means that in some cases, the algorithm crawls through hundreds of pages from one domain, while in other cases, it only retrieves a few pages. This could be partly remedied in the future by modifying the crawler so that it doesn't abandon the domain until it saves a fixed number of pages or exhausts all links. However, in practice, it often happens either that the crawler is regardless blocked by the website after making a number of requests which is deemed suspicious, or that the domain only contains a small number of pages anyway.

4.4 Alternative approaches

In order to explore strategies that are different from the proposed pipeline, we may substitute parts of the pipeline, or the entire pipeline, in the following ways:

- Using **prompt engineering**. In particular, we might feed the extracted text as a prompt to an LLM, together with instructions asking it for a category prediction. Advanced techniques for enhancing prompts such as RAG are not directly applicable, as the documents extracted from websites are typically quite long, spanning tens of thousands of characters. This is not suitable for prompting a model, since the attention mechanism of the transformers can typically only keep track of several thousand tokens. Therefore, before feeding the document to a prompt, its text must be made more concise while preserving as much relevant information as possible.
- Using **LLM embeddings** for classification. This basically amounts to replacing the embedding model with an LLM and testing the quality of the embeddings.
- Using **existing knowledge bases**. In order to do this, we will need to match the categories and businesses onto resources present in existing knowledge bases. The optimal use case would be to transfer the knowledge into our knowledge base directly, in case both the business and the category are present in the source knowledge base. If this is not viable, we might use the annotations and descriptions from the source to help with describing the category hierarchy, e.g. for the prompt engineering example above.

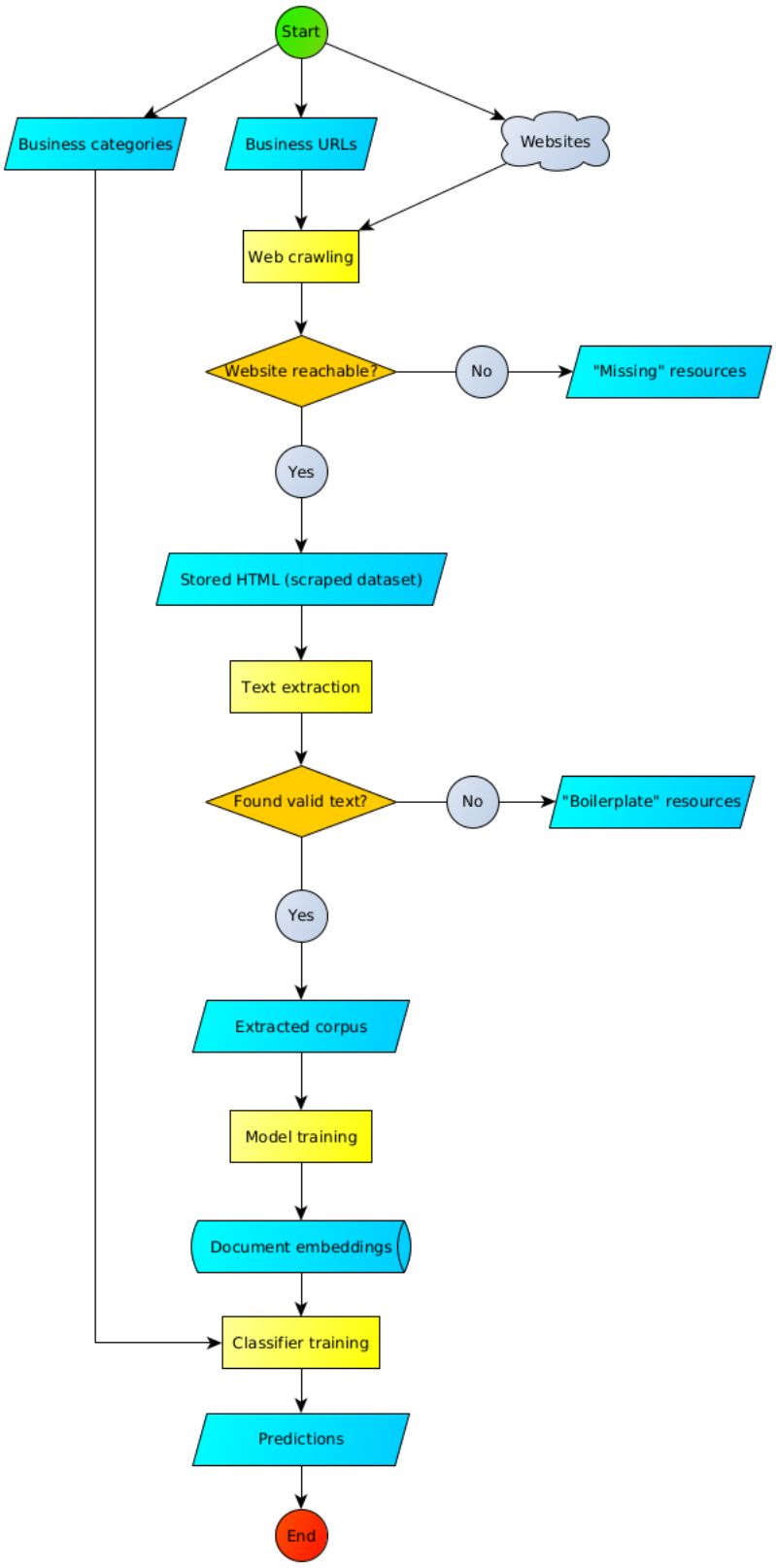


Figure 4.1: A flowchart of the classification experiment pipeline.

Chapter 5

Experiments

Unless specified otherwise, all experiments were run on a PC with the following specifications:

- OS: Debian GNU/Linux 11 (bullseye) x86_64
- CPU: AMD Ryzen 5 5600X (12) 3.7GHz
- GPU: NVIDIA GeForce GTX 980 Ti

5.1 Implementation

Three datasets were created from scraped data, each in a different target language. An English-language dataset was created using businesses operating in the UK. Similarly, a German-language dataset was compiled from businesses operating in Germany. For the Spanish language, the dataset includes businesses across several Spanish-speaking countries.

The businesses are divided into 25 categories, which are heavily imbalanced. In general, most of the analysis was done on the English and German datasets, because there were some issues with the Spanish dataset, which will be discussed later. See Appendix B for properties of the retrieved datasets and categories.

The crawlers were implemented using Scrapy. A sample Scrapy spider can be found as `crawler.py` in the attachments. The spiders for all datasets were essentially the same, only differing in the input data. The spider can be executed by creating a Scrapy project (`scrapy startproject project_name`), moving the `crawler.py` file into the `spiders` directory of the project, and then running `scrapy crawl crawler`. The `settings.py` of the project can be in the default state, though there are some changes to the settings that should be considered:

- `DEFAULT_REQUEST_HEADERS` – changing the `Accept-Language` header to reflect the target language.
- `AUTOTHROTTLE_ENABLED` – set to `True` to prevent the crawler from getting blocked for sending too many requests at once.

- `SCHEDULER_PRIORITY_QUEUE`
  – set to `scrapy.pqueues.DownloaderAwarePriorityQueue` to help the crawler not get stuck on domains that respond very slowly.

- `DUPEFILTER_CLASS` – set to `scrapy.dupefilters.BaseDupeFilter` to disable filtering duplicate requests, as this is unnecessary for this purpose and it slows the crawler down when it has already visited a large number of pages.

The crawler uses a custom package for storing the data into a defined folder. This package will be described in Subsection 5.1.1.

I also tried Scrapy with the Playwright extension, unfortunately the Playwright engine was not reliable and didn't finish crawling the entire dataset.

5.1.1 Pipeline management

No integrated MLOps environment like MLflow has been used. This allows more control over the experiments, though it also means that building the pipeline takes a bit more effort and caution. After some experiments with storing data in ways that were found to be impractical, a simple package for managing the scraped datasets was created. The package was written in Python 3.9.2 and it has no external dependencies except jusText, which is the default boilerplate removal tool. It can be found in the attachments in the `sterojos_svp_scraping_tools` folder.

The package requires the user to specify a folder where the dataset is saved. It is assumed that a folder contains only one dataset. Examples of using the package can be found in the readme file and in the experiment examples. The data is stored in zip archives which are located in the folder. Gzip was also tested for storing HTML, with zip appearing to be the better option since it allows for easy creation of archives that can contain a large number of files.

Internally, when a page is added to the dataset, it is first written into an in-memory buffer. Once this buffer exceeds a user-defined maximum number of pages, the entire buffer is written into a zip archive and emptied. However, it was discovered experimentally that some websites send HTML files that are so tremendous that they will cause a normally reasonable buffer size to explode in memory. For this reason, the memory size of the buffer is also estimated and the buffer is saved if the memory size exceeds a threshold, even if the maximum buffer size has not been reached.

Inside the dataset folder, the pipeline creates two subfolders, which store the extracted corpora and trained models, respectively. The extracted corpora are much smaller than the original HTML datasets, so they are stored as single files.

The list below contains a summary of the functionality of the classes included in the package. The first three are especially interesting for storing data and building models that interact with the pipeline.

- `ScrapeDatasetWrapper` – Implements both reading from and writing into a scraped dataset. It takes in a folder path as one of its input

arguments, assuming that the provided folder contains only one dataset. It can also read the archives and retrieve raw HTML for a given URL or domain. The output files are written into zip archives with fixed-length random strings as file names. Additionally, each of the produced archives contains a CSV file that maps the URLs to the file names.

- `TextClassificationPipeline` – Feeds the data from a dataset defined by the wrapper to a model and trains the model. Internally, it uses a Python context manager to ensure that the model and the dataset are deleted after the pipeline finishes.
- `TextClassificationModel` – An abstract class defining the interface that must a model must implement to interact with `TextClassificationPipeline`.
- `ScrapeTextExtractor` – Connects the dataset with input merchants and categories, retrieves text for a given merchant using a boilerplate remover (**jusText** by default), determines the success rate of the scraping and detects suspicious cases.
- `BaseSequentialModel` – Implements some of the functionality of the abstract `TextClassificationModel`, assuming that the phases of the model form a simple sequence.
- `PipelineMetadataHandler` – Saves data about already existing models into a JSON file located in the `__models` subfolder of the dataset folder. The data can be later "queried" using patterns that the model must satisfy (type, parameters, used extractor etc.) using the `find_models` method which functions similarly to the MongoDB `find` command.

The script `example_experiment.py` in the attachments demonstrates the integration of a model into the pipeline code on the example of classifier using TFIDF + Naïve Bayes. This example should be runnable on the attached artificial data. It should also be possible to run the Top2Vec pipeline using `top2vec_experiment.py`, though this requires the modified Top2Vec package.

The implementation of `BaseSequentialModel` uses string names to identify the phases and requires the user to manually specify which model parameters influence which phases. This was done to simplify the integration with existing modifications to the Top2Vec code, but it could have been done in a cleaner way, e.g. by defining a decorator for a phase function and automatically collecting the parameters of decorated functions using Python's `inspect` module.

5.1.2 Testing the implementation

When the pipeline is initialized, it first verifies that all of its components (including the model) can be initialized, and whether the model can be loaded from an existing file to skip some phases. However, this assumes that the

implementation of the pipeline itself is correct, i.e. it can gauge the errors and perform all operations properly.

Given that the pipeline contains several components that take a longer time to complete on a real dataset, it is crucial to first create tests verifying that all components are functioning properly and that they can be integrated into the pipeline successfully.

Testing the pipeline can be a challenging task, as its execution depends on the properties of the dataset. In particular, the text extraction, which is the first step in the pipeline, can potentially discard the entire dataset if it does not seem like valid data. This means that if the testing dataset is designed incorrectly, it is not possible to test the later steps of the pipeline.

The module `utils.py` in the package contains functions that generate random text documents and faux HTML datasets. These functions are designed with the following goals in mind:

- The generated text should be "similar enough" to some natural language to fool an extractor into not marking them as boilerplate. (The provided functions can bypass boilerplate removal by the jusText extractor by being sufficiently similar to English-language text.)
- The documents should have a basic HTML structure, so that the ability of the extractor to transform HTML documents into a text corpus is verified.
- The documents should be tagged with "URLs" like in a real scraped dataset.
- The generated text should have various "topics", where each segment (paragraph) of the text is generated from the vocabulary of a single topic.
- The topics that are used in a particular document should be determined by the domain of the document's URL. This ensures that the entire domain can be assigned to a particular topic (which is what we are trying to accomplish with real websites).

Using these functions and the `unittest` module from the Python standard library, several tests were implemented. The tests target the basic functionality of the pipeline components and the pipeline itself, and can be found in the `tests` folder in the attachments.

5.1.3 Proposed modification to the Top2Vec package

Top2Vec provides a unified and consistent interface to discovering topics from corpora, building upon Gensim's implementation of Doc2Vec, UMAP, and TensorFlow's repository of pre-trained models. For this reason, we have decided to build upon Top2Vec's solution to the problem, rather than create a new one from scratch. However, some issues with the package have been identified:

- The package doesn't treat internal variables consistently – some (e.g. high-dimensional document vectors) are stored as the object's attribute and can thus be loaded from a saved instance, while others (e.g. low-dimensional vectors produced by dimension reduction) only have a local scope, despite being less memory-consuming in some cases.
- The `Top2Vec` object has no interface to only run a part of the algorithm, making it effectively impossible to use a loaded instance to skip the earlier phases, or to save an unfinished model for future use. The entire algorithm runs inside the constructor of the `Top2Vec` class and isn't structured in a way that would allow to re-run a part of the algorithm e.g. by calling a method of the object. This is quite limiting, since the preprocessing the documents and generating embeddings can take hours, and a new model that only tweaks a downstream operation cannot be created without first re-running the earlier, expensive steps.

To improve the utilization of the package during the experiments, it was modified in the following ways:

- The `Top2Vec` class was modified to save almost all variables as attributes to the instance of the object. This was done partly to be able to recover the values of these variables from a saved model, but also because the model has many parameters and by saving these parameters as attributes, we avoid having to pass them as parameters to internal functions later.
- Identifiable "phases" of the algorithm (e.g. tokenization, discovering topics...) were moved to separate functions whenever this wouldn't require drastic changes to the structure of the class itself.
- The default parameters were moved to a config file `Top2Vec.yaml`.
- A new class method `Top2Vec.run()` was created. This method generalizes the existing constructor of the `Top2Vec` class and could potentially replace it. The most important change is that besides being able to run from a corpus, the model can also build upon an existing instance, provided that the parameters match.

Top2Vec doesn't provide support for dimension reduction methods other than UMAP, but this can be amended quite easily. The ability to use PaCMAP instead of UMAP was also added to the package. Switching between PaCMAP and UMAP can be done using the `use_pacmap` parameter, which is `False` by default.

All changes made to the package can be found in the attached diff file `Top2Vec.diff`. Alternatively, the modified package can be retrieved from a GitHub fork of the package[1]. The name of the package was changed to `top2vec_modified`, so that it can be installed alongside the official version of the package.

[1] https://github.com/sterojos/Top2Vec – The attached diff was generated by comparing commits 7403993 and eb3cf3e. The source top2vec version is 1.0.34

Top2Vec was integrated into the code of the pipeline with the attached file `top2vec_experiment.py`, in particular the `Top2VecModel` class.

5.2 Results

This section discusses the results with respect to the feasibility of using the embeddings for classifying the merchants into the pre-defined categories. Results from the alternative approaches to classification can be found in Section 5.4. See appendix C for more figures illustrating the properties of the datasets.

5.2.1 Dataset properties

In the early experiments, three main approaches to the dimension reduction were tested on the UK dataset:

- using a traditional bag-of-words representation and a Bayesian model, i.e. latent Dirichlet allocation,
- using embeddings from a pretrained model – Universal Sentence Encoder from Top2Vec,
- training a custom Doc2Vec model of the dataset using Top2Vec.

In all cases, the jusText extractor was used to convert the HTML dataset into a corpus.

The success of the classification is directly dependent on properties of the manifold of documents resulting from the dimension reduction of the document embeddings. Figure 5.1 shows an example of a reasonably good result acquired from Latent Dirichlet allocation (LDiA) embedding of the UK corpus. Compare to Figure 5.2, which shows a PCA projection of Doc2Vec embeddings of this corpus. Linear methods generally cannot capture the complexity of the data in lower dimensions. The combination which achieved best classification results was the Trafilatura extraction on the German-language dataset.

It was found that manifold learning methods can also help to increase separation between clusters in the data. In figure 5.3, nearest neighbors in the high-dimensional space are connected by a line in the PaCMAP-reduced space. It is apparent that PaCMAP did a lot of work to "unravel" the manifold into a shape that preserves its local properties. However, this does not mean that the dataset is distorted, as such extreme transformations are only done when necessary and this is caused by the high noise in the data. When the nearest neighbor graph is constructed on an artifically separated dataset, it's visible that the separation between the clusters is preserved.

The jusText extraction appeared rather error-prone at first, seemingly including script and JSON files in the extracted text. From the 20 most common bigrams in the extracted corpus, only 2 could plausibly appear so

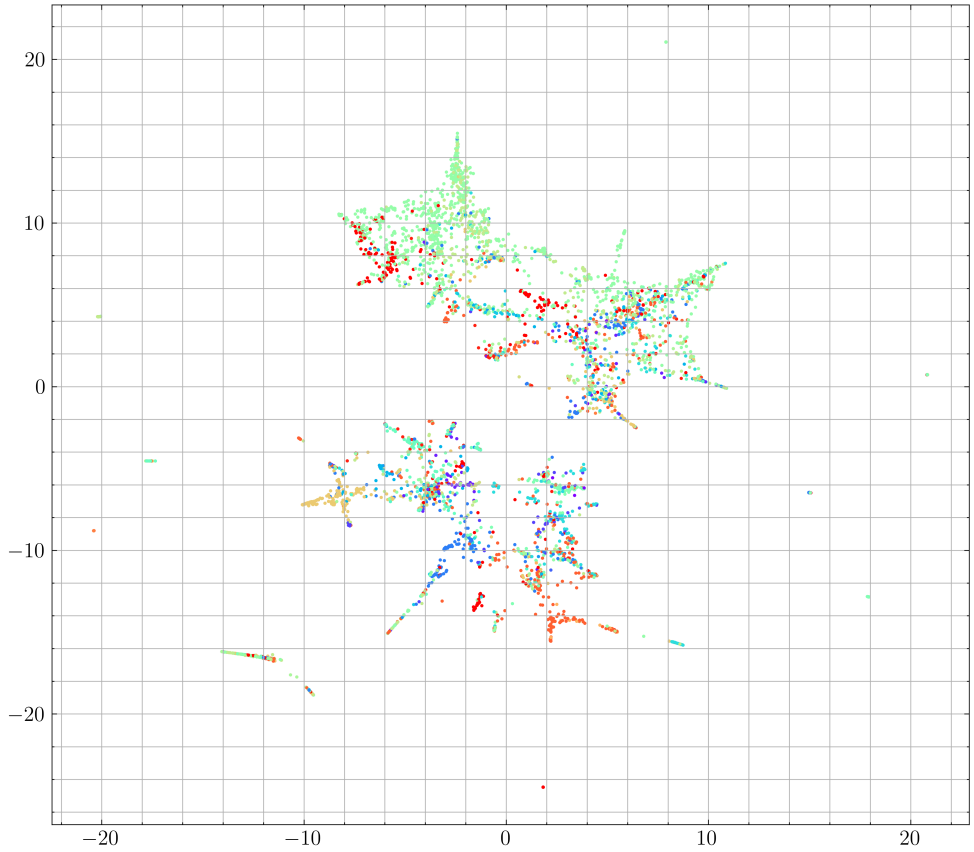


Figure 5.1: PaCMAP projection of LDiA document embedding of the UK dataset, extracted using jusText. The points are colored according to the 25 distinct categories.

frequently in text – (“,”, “and”) and (“of”, “the”), while the rest seemed to come from structured files – e.g. (“:”, “{”) or (“matchLevel”, “´´”). Searching for suspicious keywords such as `matchLevel`, `matchedWords` in the corpus reveals that they come exclusively from the `prada.com` domain. It appears that these keywords come from a JSON data file sent by `prada.com`, which is unusually enclosed in a `<pre>` tag. jusText assumes that a long `<pre>` tag with a sufficient ratio of stopwords is not boilerplate, and the JSON file also includes nested text which probably contains enough stopwords. The domain `prada.com` was omitted from the set of successfully scraped domains, but it was found that this did not significantly affect the quality of the embeddings. Later, it was also verified that the corpora obtained using different extractors (trafilatura, boilerpy3) also contains the bad data from `prada.com`, and in the German dataset, a similar inspection of bigrams didn’t reveal anything suspicious.

Scraping was tested on UK businesses, where it was determined that only about 50% of businesses had a unique domain that contained non-boilerplate text in the HTML. Many merchants are concentrated on a small set of domains such as `facebook.com`, `maps.google.com`, to each of which several hundreds

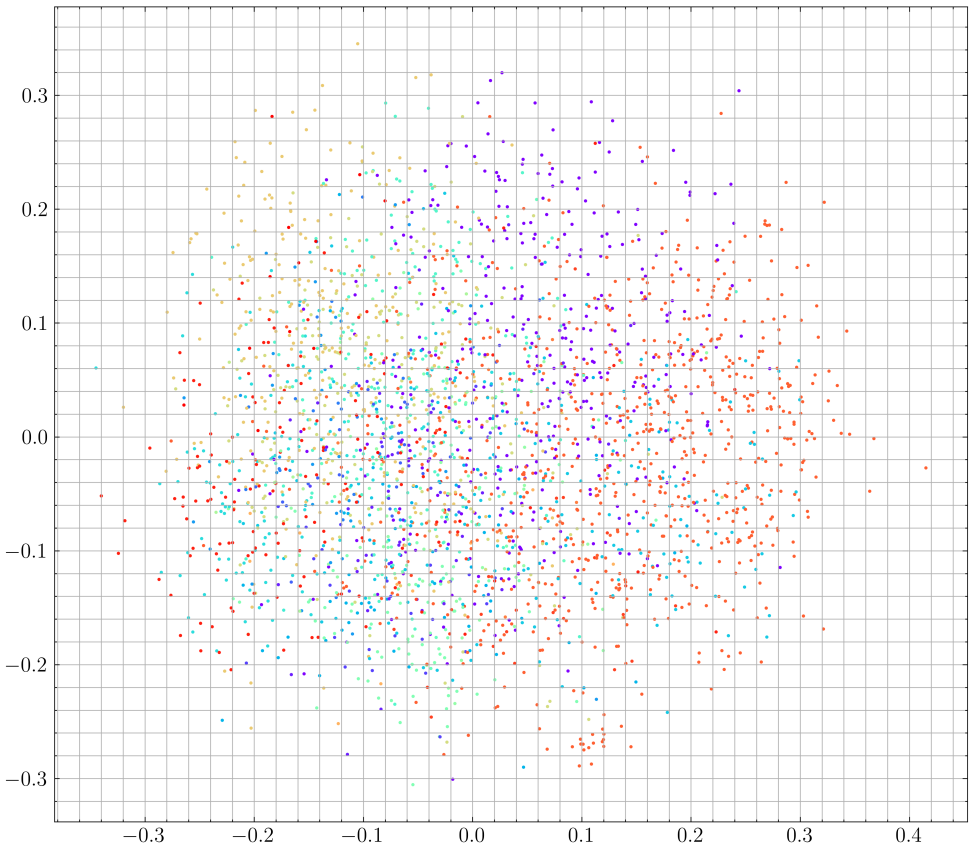


Figure 5.2: PCA projection of Doc2Vec document embedding of the UK dataset, extracted using jusText. The points are once again colored by category.

of merchants are linked. For now these merchants will be ignored. Moving forward, a different approach will need to be employed for these domains.

Scraping inexplicably failed for about 500 domains, most of which appear to be normally accessible. This might be due to a missing header or cookies. Unfortunately, it is slow to keep track of cookies when crawling a large number of domains.

For many websites, the connection was refused due to missing headers. This could probably be avoided by using different headers imitating a human user.

On the Spanish dataset, Top2Vec with Universal Sentence Encoder failed to converge unless results from Catalonia (top-level domain `.cat`) were excluded. This is probably because the local businesses use the Catalan language instead of Spanish, which is similar enough to Spanish for the boilerplate remover to accept it as Spanish text, but distinct in more advanced vocabulary which the Universal Sentence Encoder does not recognize. Generally, the quality of the Spanish data was lower and the dataset was smaller. Some embeddings of the Spanish corpus can be found in Appendix C together with other visualizations.

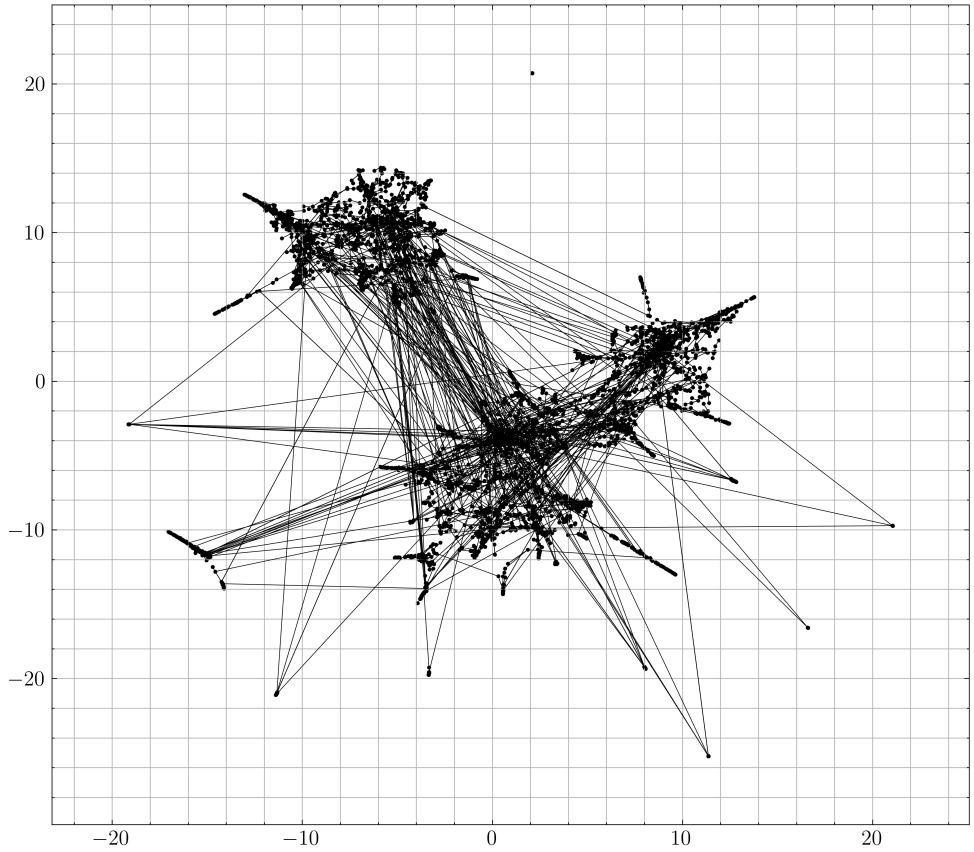


Figure 5.3: Visualization of the dimension reduction performed by PaCMAP on the LDiA document embedding of the UK dataset. Each point is connected with a line to the point which was its nearest neighbor prior to the dimension reduction.

5.2.2 Comparison of classifiers and extractors

Tables 5.1 and 5.2 show the results of different classifiers and extractors on standard Top2Vec embeddings (Doc2Vec + UMAP with 5 output dimensions). In both cases 70 % of the dataset was used as training set and 30 % as test set.

In the tables, MF1 stands for micro-averaged F1 score and WF1 is the weighted average F1 score. The weighted F1 score is usually very similar to the accuracy, as is evident from the German data. For this reason, it is not reported for the English data.

As part of the early experiments, I tried to use train a feedforward neural network to classify the high-dimensional LDiA embeddings of the jusText-extracted UK dataset (depicted after reduction in Figure 5.1). There was an issue with overfitting on the training data, but I was able to mitigate this by adding dropout to the network, in the end reaching an accuracy of 63.94 %. At the time I believed that this was a bad result, but seeing the performance of Top2Vec embeddings, the issue was clearly in data quality,

	jusText			Trafilatura		
Classifier	ACC	MF1	WF1	ACC	MF1	WF1
LDA	48.6	23.0	44.0	60.3	31.1	56.3
Logistic regression (LBFGS)	49.8	24.8	45.7	62.7	34.3	59.2
Logistic regression (SAG)	53.8	30.2	49.9	62.9	34.1	59.4
Random forest	66.8	48.7	66.1	69.9	55.6	69.5
RBF-kernel SVM	61.7	36.4	59.3	63.4	32.3	59.5
5-NN	**66.7**	**52.0**	**66.4**	**70.0**	**57.8**	**69.7**
3-NN	65.9	49.7	65.8	67.5	53.6	67.5
TF-IDF + Naive Bayes	24.1	0.02	0.10	0.26	0.03	0.13

Table 5.1: Results for various classifiers on the German dataset, using a custom Doc2Vec model and the UMAP dimension reduction.

	jusText		boilerpy3		Trafilatura	
Classifier	ACC	MF1	ACC	MF1	ACC	MF1
LDA	56.8	31.1	58.1	32.2	57.9	35.2
Logistic regression (LBFGS)	59.0	34.8	56.2	28.9	58.3	31.5
Logistic regression (SAG)	59.3	36.1	56.4	29.5	59.6	32.2
Random forest	**66.2**	46.7	65.5	45.7	66.5	**49.8**
RBF-kernel SVM	61.8	35.7	60.1	28.3	63.7	37.7
5-NN	65.8	**49.1**	**65.8**	**46.5**	**66.6**	49.5
3-NN	62.7	46.8	62.7	45.0	63.5	45.3

Table 5.2: Results for various classifiers on the English dataset, using a custom Doc2Vec model and the UMAP dimension reduction.

and this might be a viable alternative – both LDiA and a simple feedforward network consume much less time than Doc2Vec. The training can be found in a Jupyter notebook in the attachments as `ldia_nn_training.ipynb`.

5.3 Discussion

In an ideal world, much of the work done in this thesis would not be needed, as web developers would be able to use already established ontologies such as Schema.org or OpenGraph to embed the relevant structured data directly into the source code of their websites. However, even though the use of these ontologies is quite widespread, it is not nearly frequent enough for the construction of a knowledge base covering the businesses in a given country.

The embedding space tends to be remarkably complex regardless of the employed language model. This has several consequences:

- Clusters discovered in the data by algorithms such as HDBSCAN do not directly correspond to categories, though some of them might represent subsets of categories. This will be elaborated upon in Subsection 5.4.3.
- Linear algorithms fail to learn anything useful from the data. In par-

ticular, algorithms like PCA can't find helpful subspaces, and linear classification algorithms typically don't converge.

- Classifiers allowing more complexity, such as kernel SVMs and neural networks, easily overfit on the training data.

Nonetheless, the categories were deliberately not simplified to investigate the possibility of fully automating the categorization process using supervised learning from already categorized businesses. Note that the human labels, while verified, can also contain some mistakes.

In a practical setting, since K-NN performs well on this data, classification might be reasonably reliable using the following classification process:

- For each document in the test set, retrieve K (e.g. 3) documents from the training set which are the nearest neighbors.
- Return a category prediction $\mathcal{C}$ if all K neighbors have class $\mathcal{C}$. If the categories of the neighbors don't agree, return "unknown".

This way, we might maximize the precision of the classification, sacrificing some accuracy. Since we are looking for knowledge about the websites, it's more important to be precise than to cover all cases.

5.4 Alternative approaches

I also tried some of the experiments outlined in Section 4.4. In this section, I will describe the limitations and results.

5.4.1 Extracting tag descriptions from existing knowledge bases

Two open knowledge bases that contain structured general knowledge were investigated for this purpose, Wikidata[2] and DBpedia[3]. While DBpedia is a project based on automated extraction of knowledge from Wikipedia articles, Wikidata is a knowledge base that can be linked to Wikipedia, but is a separate project that contains a mixture of procedurally generated and human input.

The idea is to match our categories onto Wikidata resources, and use the descriptions provided with these resources to help create a more specific prompt or sequence for embedding. The motivation is that models often struggle to distinguish between some categories where the difference is not immediately obvious in all cases, such as "Groceries" and "Food And Drink".

However, experiments showed that this method of creating category descriptions is unreliable, not because of being wrong, but because of being incomplete. A simple client, `wikidata_fetcher.py` (found in the attachments), was made to retrieve descriptions of categories from Wikidata. Some

[2] https://query.wikidata.org/
[3] https://dbpedia.org/sparql

Wikidata ID	label	description
Q180846	supermarket	large form of the traditional grocery store
Q864440	health food store	type of grocery store that primarily sells health foods, organic foods and local produce
Q2024419	online grocer	e-commerce service that sells retail foodstuffs and other household supplies
Q7361709	convenience store	small store that stocks a range of everyday items
Q11914717	colmado	type of grocery store specific to Barcelona
Q27676067	Amazon Go	chain of grocery stores operated by Amazon.com
Q107452610	grocery store bus	bus used as a mobile grocery store
Q111593675	social supermarket	

Table 5.3: Direct subclasses of "grocery store" in the Wikidata knowledge base as of March 2024.

top-level categories were manually matched onto closest Wikidata equivalents. Note that this experiment was done in March 2023 and the Wikidata knowledge base changes dynamically, much like Wikipedia.

Table 5.3 shows an example of resources that are considered direct subclasses of a grocery store in Wikidata. Note that while some of these might be useful to explain what different types of businesses are considered grocery stores, others ("store in Barcelona") might actually be counterproductive. Some other examples were even worse – either the generic concept barely had any subclasses and businesses attached to it, or many of them refer to very specific concepts particular to a location or culture.

DBpedia reliably extracts basic information about a given subject, but doesn't process the content of the articles very deeply. Contrary to initial expectations, it seems that DBpedia might actually be the better candidate for a reliable knowledge base to extract data from. Compared to Wikidata, it has much better annotations and correct tags for companies, although the ontology doesn't always work in an expected way. Utilizing an external knowledge base with a more complex ontology leads to a more complex solution of the problem of entity matching, which is out of the scope of this thesis.

5.4.2 Using LLMs for text classification

I attempted to use large pretrained models for text classification. There are essentially two straightforward ways this could be done: either by means of prompting a model with instructions and, hopefully, retreiving the prediction from a sensibly formatted answer, or by extracting embeddings of documents from the model, and then leveraging the meaning encoded in the embedding

space to perform the classification itself using a simpler model.

I ran some LLMs locally to see how much they are capable of, and also to what extent such tasks are computationally manageable. I used 7B models, which are typically the "smallest LLMs" available, so the results might not be entirely relevant for larger models.

Text classification by prompting

In order to fit the most relevant text from each document in the corpus into a prompt, I used TF-IDF to create scores for words, and then selected the sentences from the text which had the highest average TF-IDF score across all words which are not stopwords (i.e. very common words like *is*, *he*). There are issues with this approach, since can alter the ordering and disturb the overall flow of the text. However, text gathered from business websites rarely tells a coherent story.

I tested two models locally in this way, namely LLaMA 2 and Mistral, via Ollama. The models were tasked to classify the text into one of the predefined categories and were informed that the text comes from the website of the business and that they ought to use it to help them make their decision. The experiments can be found in the `prompting` folder in the attachments.

One positive outcome of this experiment is that both models understood the assignment and didn't lose track of the context, despite being given the instructions at the very beginning of the lengthy prompts.

A minor issue is that they tended to sometimes respond with short sentences when they were told to only reply with the category name, but their responses were predictable enough that the category name could very likely be extracted with a regular expression. A worse problem with the responses is that they sometimes were invalid, i.e. they contained a hallucinated category not provided as an option.

Unfortunately, the quality of the LLaMA 2 responses was very low and could even be influenced by changing the category which was given as an example of a "correct response" (the model responded to the example by being significantly biased towards this category). The Mistral model appeared to understand the task better, but the accuracy of its responses was still quite low.

Note that this exercise shouldn't be understood as portraying poor reasoning skills by the models. The categorization is sometimes not entirely obvious, and the models were only given a list of category names without any clarification. When the businesses are categorized by humans, they are given prior training that aims to disambiguate some common edge cases to preserve the regularity in the category system.

On a related note, the Perplexity service seemed like a good tool for the task, giving reasonable predictions when presented only with a list of categories and a URL[4]. For this reason, I also tested the Perplexity AI Sonar model,

[4]For instance, the Cider clothes store was classified correctly despite the confusing name in this conversation.

available through an API, via the Perplexity Labs[5]. Since this model should have access to the Internet, I only provided the links to the websites to see if the model can gather the information itself. However, the model got lost in the conversation often and had similar issues to LLaMA2. That being said, there is now a LLaMA 3-based model in the same API which might work better.

Using LLM embeddings for text classification

Since Mistral seemed to be a reasonably good model for the task based on the prompting experiment, I also tried to extract embeddings of the documents from the model and use it just like the other embedding models. Unfortunately, the embeddings were found to be of worse quality than the Doc2Vec embeddings, comparing using K-NN classifier after dimension reduction (this only reached about 40% on Mistral embeddings). The Mistral embeddings can be seen visualized in the usual manner in the appendix. They can be generated via Ollama like this:

```
from langchain_community.embeddings import OllamaEmbeddings

documents: list[str] = ["doc 1", "doc 2"]

embedder = OllamaEmbeddings(model="mistral")
embeddings = embedder.embed_documents(documents)
```

Later, I tried to acquire document embeddings on the English dataset from LLaMA 3, seeing that it is now the state-of-the-art model for English. Unfortunately, the GPU used for the experiments ceased to function during the generation of these embeddings, and was therefore no longer available for further experiments.

5.4.3 Using clustering to categorize businesses

The aim of this experiment was to explore how the human-made category labels are related to the topics obtained using Top2Vec (i.e. HDBSCAN). First, the Jaccard index J was computed for all topic-category pairs across the dataset:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.1}$$

Then, for each topic, a best category match was found as the category with the highest Jaccard score with respect to the topic. Analogously, a best topic match was found for each category.

Then, for each document, it was evaluated whether this document is matched "as expected", i.e. the topic of the document is the topic match for the category of the document, and the category of the document is the category match for the document's topic. A visualization of this in the

[5] https://labs.perplexity.ai/

PaCMAP-reduced space can be found in the appendix. This relationship would perhaps be better visualized as a bipartite graph, but this would be challenging to depict properly and fit onto a page since the topics outnumber the categories by 62 to 25.

More importantly, we should try to answer the question of whether it is possible to use the topics to assign the documents to a category, since this is the way one might practically utilize the topics to accelerate the assignment of categories to a large corpus of documents. Table 5.4 is an attempt to answer this question. For a selection of large topics (i.e. topics that contain many documents), the similarity to the closest category was evaluated as the fraction of documents in the topic that belong to the closest category. The most similar words to the topic were retrieved from Top2Vec to illustrate the meaning of that topic.

From Table 5.4, it appears that if there is a good category match, the words most similar to that topic match the category quite well, even if this is not immediately obvious. Here, it is worthwhile to investigate the tag hierarchy which is described in Section 5.5. For example, the topic represented by words "djs, gigs, ..." might evoke a night club, not "Food And Drink"[6]. In the tag hierarchy, there is a "Night Club" tag under the "Freetime" category, but there is also a "Bar" tag under the "Food And Drink" category. In fact, the "Bar" tag has a much higher prevalence than the "Night Club" tag (9777 to 29). Therefore, we can probably attribute this topic to the "bars" present in the input data.

There are also some topics that don't clearly correspond to any category, but this can likely be attributed to the noisiness of the data. If better embeddings could be acquired, categorizing the businesses "en masse" by assigning whole topics to categories might be a helpful tool to speed up the categorization process if no data is yet available for a given country or language. However, this would also be risky, since having no input data, there would be no way to assess the quality of the embeddings.

5.5 Creating a knowledge base

To demonstrate the usage of the models' predictions in a knowledge base, I used Ontotext GraphDB, which is a graph database compliant with RDF and SPARQL standards. The tag tree, which is visualized in Figure B.1 located in the appendices, was converted into an RDF representation. This representation can be found in the `knowledge_base` folder in the attachments. Besides describing the hierarchy, the RDF files contain some extra information, such as how many merchants belong to a tag in the TapiX database, and a couple Wikidata IDs of the categories which were matched onto Wikidata resources.

[6]The reasoning behind this, as explained by a specialist, is that spending inside music clubs is dominated by drinks, so this is the correct category to assign to most payments that happened inside a music club.

Topic size	Most similar words	Most similar category	Similarity
215	ales, beers, lagers, beer, pub	Food And Drink	0.930
202	timber, decking, cladding, flooring, roofing	House And Garden	0.870
201	djs, gigs, anthems, music, dancefloor	Food And Drink	0.789
159	hotel, rooms, overlooking, ensuite, guests	Travel	0.596
149	bakers, cakes, cake, patisserie, baked	Groceries	0.505
147	wardrobe, outfit, jeans, flattering, fatface	Fashion	0.725
134	brasserie, dishes, menus, ramsay, restaurant	Food And Drink	0.934
126	flapper, pm, am, ormsby, sat	Food And Drink	0.541
118	recruitment, candidates, recruit, candidate, roles	Professional Services	0.853
105	noodles, rice, curry, tofu, spices	Groceries	0.479
95	automate, workflow, spreadsheets, zoho, pipedrive	Digital Services	0.627
95	smugmug, foregoing, hellofresh, therein, flickr	Food And Drink	0.262
86	governance, leadership, glanbia, stakeholders, shareholders	Food And Drink	0.234

Table 5.4: Examples of large topics discovered by Top2Vec and their closest category matches.

We will also need to convert the outputs of the pipeline into RDF. The pipeline stores the models' predictions into the `__models` subfolder of the dataset folder, where they are stored in a compressed CSV file together with the serialized model. These predictions can be converted to RDF using the provided example script, `prediction_to_rdf.py`, as follows:

```
python3 prediction_to_rdf.py -i results.csv.gz -o merchants.ttl
```

Thus the raw data from the predictions is correctly included into the RDF graph of the tag tree. The script requires the `rdflib` Python library, which was also used for all other RDF processing.

I created a new repository in GraphDB (Figure 5.4) and inserted all of the RDF files into this repository Figure 5.5. Then, I ran an example SPARQL query in the repository, which can be found in Appendix D. This query estimates the most likely number of merchants in the repository that should have a given tag, using the relative frequency of the tags in the given categories according to the counts included in the tag tree, and the number of merchants with a given category in the extracted repository. Then, the tags are sorted by this expected number of occurences, so that the tags that are expected to

be most frequent appear on top. The result of the query is in Figure 5.6.

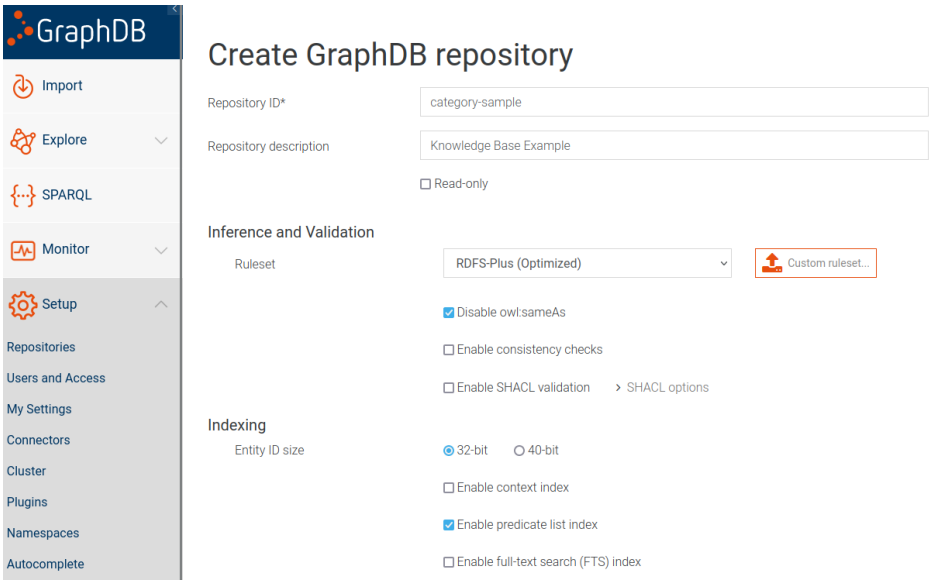


Figure 5.4: Creating a repository in GraphDB.

5.6 Further steps

I believe the solution of data storage and model training is satisfactory for now. Moving forward, what would probably require the most work is the text extraction, because the existing solutions are not satisfactory even for (mostly) small websites that this was tested on, judging by the accuracy achieved by the classifiers when comparing e.g. to [RBP24]. The issue is that this is also the part of the process that is the least covered by existing research, so it's difficult to determine what could improve the performance reliably. Replacing the plain HTML crawler with a headless browser might not be a top priority, since in this specific case the content we're looking for is usually not dynamically generated (websites that were scraped successfully, but had no valid content were rare).

As far as further development of the package is concerned, below are some encountered issues and ideas for improvement.

- Text extraction is not parallelized by default. This could be amended using Python's native `multiprocessing` library. However, compared to the other parts of the pipeline such as training Doc2Vec models, text extraction is a relatively fast process.

- The URLs and categories supplied to the extractor have to have a very rigid structure (CSV file with gzip compression and correct order of columns).

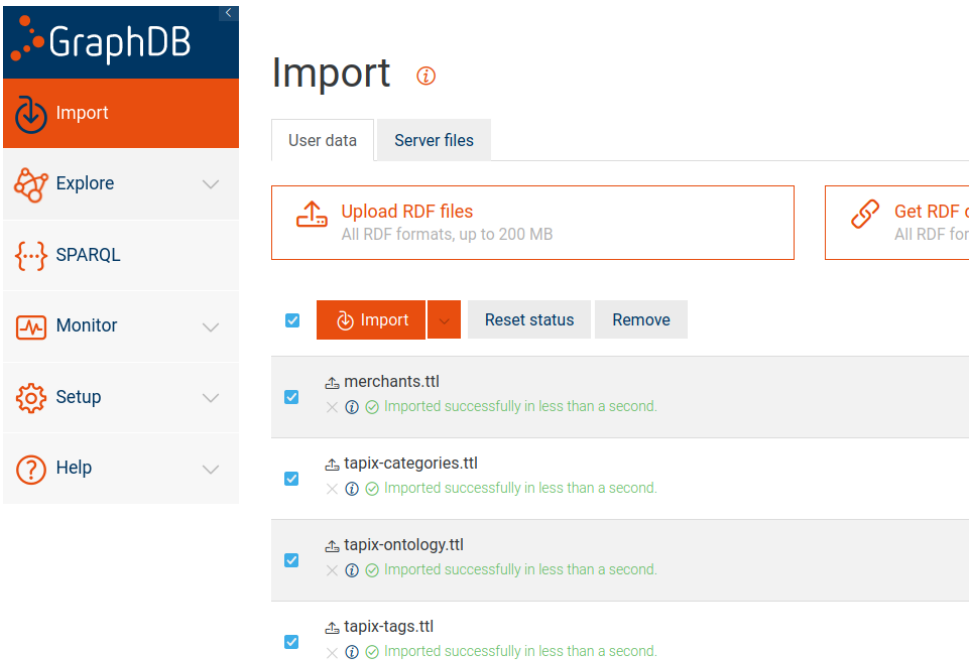


Figure 5.5: Importing Turtle files into GraphDB.

- Lambda expressions cannot be used as the extractor, because the name of a lambda function is internally `<lambda>` instead of the name of the variable in which it is stored.
- Since the package uses the name of the module to identify the functions/model classes, these have to be imported from a module with a specific enough name. If these components are defined directly in the script running the pipeline, `__main__` is used as the module name, which could lead to confusion.
- The package doesn't utilize any databases, instead using file hierarchies and ZIP archives for storage. This was not an issue as far as the experiments for this thesis are concerned, but it could nonetheless be problematic when scaling to significantly larger datasets.
- Python's native `csv` exporter doesn't seem to be entirely reliable, sometimes apparently inserting NUL bytes into the files that make the files impossible to decode for the native CSV reader (while other CSV readers can still read them).

Filter query results

Showing results from 1 to 420 of 420. Query took 0.1s, minutes ago.

	category	tag	expected_tag_count
1	txc:FoodAndDrink	txt:Restaurant	"64.52909093264584331964153 0"^^xsd:decimal
2	txc:Groceries	txt:GroceriesOther	"41.69896270139042154049377 0"^^xsd:decimal
3	txc:FoodAndDrink	txt:Drinking	"39.04354247366852789905148 5"^^xsd:decimal
4	txc:FoodAndDrink	txt:Bar	"36.98563266786718314785644 5"^^xsd:decimal
5	txc:Travel	txt:Accommodations	"29.67801442227066912627687 0"^^xsd:decimal
6	txc:Travel	txt:HotelsAndGuesthouses	"27.93325507294985745429978 6"^^xsd:decimal
7	txc:FoodAndDrink	txt:Cafe	"19.61066991410693233491744 0"^^xsd:decimal
8	txc:Car	txt:GasStation	"15.18276841891617617211394 3"^^xsd:decimal
9	txc:HouseAndGarden	txt:HouseholdEquipment	"14.95564747788119039411724 2"^^xsd:decimal
10	txc:ProfessionalServices	txt:ServicesMix	"14.46661889406425415024531 8"^^xsd:decimal

Figure 5.6: Result of the GraphDB query.

Chapter 6

Conclusion

In this thesis, we investigated multiclass text classification on corpora extracted from websites of businesses. A Python package was created for storing and managing large HTML datasets. Using the package, datasets were created from scraped websites. Web scraping and HTML boilerplate removal was applied to several thousands of websites. The datasets were analyzed with respect to pre-determined ground truth categories in the following way:

- Embeddings of the text from the websites were obtained using various language models.
- Quality of the embeddings was assessed by training classifiers on the pre-determined categories.
- In the embedding space, categories were compared to groups of documents discovered using clustering.

The Top2Vec package was modified to allow running from an existing instance to save time when training many classifiers on the same model.

Based on the findings of this thesis, key features of the solutions to the problem that should be prioritized include:

- Explainability: Using manifold learning methods is extremely helpful for visualizing the embeddings. The quality of the model can be assessed visually, which can reveal more about the model than a simple classification score.
- Efficiency: Training a Doc2Vec model is quite fast even on a CPU, and is surprisingly effective on noisy datasets. In comparison, even a large model can produce bad embeddings, when the documents are not preprocessed properly.
- Tailor-made models: Even simple models such as latent Dirichlet allocation seem to have the potential to be equivalently effective to pre-trained models.
- Dataset preprocessing: Text extracted from many different websites is unpredictable and noisy when compared e.g. to datasets of news articles.

This means that simple techniques such as TF-IDF fail to process the text. However, those same simple techniques could potentially be used to analyze the dataset and improve its quality.

While customizing the solutions for this particular problem seems to be the best path forward, making too many assumptions about the datasets should be avoided[1].

In the future, extracting other information from the websites might also be of interest, e.g. brand name, name of the legal entity, or locations of physical points of sale.

[1] http://www.incompleteideas.net/IncIdeas/BitterLesson.html

Bibliography

[Ang20] Dimo Angelov. "Top2Vec: Distributed Representations of Topics". In: (2020).

[Baa+07] Franz Baader et al., eds. *The Description Logic Handbook. Theory, implementation, and applications*. Second edition. Title from publisher's bibliographic system (viewed on 05 Oct 2015). Cambridge: Cambridge University Press, 2007. 1601 pp. ISBN: 9780511711787.

[BHN23] Anas Bodor, Meriem Hnida, and Daoudi Najima. "From Development to Deployment: An Approach to MLOps Monitoring for Machine Learning Model Operationalization". In: Nov. 2023, pp. 1–7. DOI: 10.1109/SITA60746.2023.10373733.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. New York, NY: Springer Science+Business Media, LLC, 2006. 758 pp. ISBN: 8132209060.

[Blu20] Andreas Blumauer. *The knowledge graph cookbook. Recipes that work*. Ed. by Helmut Nagy. 1st edition. Wien: edition mono/monochrom, 2020. 256 pp. ISBN: 3902796707.

[BNJ03] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. "Latent Dirichlet Allocation". In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 993–1022. ISSN: 1532-4435.

[Bor23] Mitchell Borchers. "Active learning in E-Commerce Merchant Classification using Website Information". MA thesis. Charles University, Faculty of Mathematics and Physics, June 2023. URL: http://hdl.handle.net/20.500.11956/181873.

[Cer+18] Daniel Cer et al. "Universal Sentence Encoder". In: (Mar. 2018). DOI: 10.48550/ARXIV.1803.11175.

[CMS13] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. "Density-Based Clustering Based on Hierarchical Density Estimates". In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Jian Pei et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–172. ISBN: 978-3-642-37456-2.

[DE19] R. Kanniga Devi and G. Elizabeth Rani. "A Comparative Study on Handwritten Digit Recognizer using Machine Learning Technique". In: *2019 IEEE International Conference on Clean Energy and Energy Efficient Electronics Circuit for Sustainable Development (INCCES)*. 2019, pp. 1–5. DOI: 10.1109/INCCES47820.2019.9167748.

[Dev+18] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: (Oct. 2018). DOI: 10.48550/ARXIV.1810.04805.

[DS23] Çağdaş Doğan and Sinan Sarıca. "Comparison of Classifiers for Text Classification in an E-commerce Service". In: *2023 14th International Conference on Electrical and Electronics Engineering (ELECO)*. 2023, pp. 1–5. DOI: 10.1109/ELECO60389.2023.10415941.

[Est+96] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[EVW16] Meng Joo Er, Rajasekar Venkatesan, and Ning Wang. "An online universal classifier for binary, multi-class and multi-label classification". In: *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, Oct. 2016. DOI: 10.1109/smc.2016.7844809.

[Fas08] Ralph W. Fasold, ed. *An Introduction to Language and Linguistics*. 1. publ., Repr. Cambridge [u.a.]: Cambridge University Press, 2008. 540 pp. ISBN: 9780521612357.

[G+23] Devaraja G et al. "A Comparative and Analytical Study of Text Classification Models using Various Metrics and Visualizations". In: *2023 OITS International Conference on Information Technology (OCIT)*. IEEE, Dec. 2023. DOI: 10.1109/ocit59427.2023.10430972.

[Gan23] S. Ganeshmoorthy. "Classification of Web Pages: A Comparison of Recent Machine Learning Techniques". In: *2023 7th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. 2023, pp. 595–602. DOI: 10.1109/ICECA58529.2023.10394818.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Gra13] Alex Graves. "Generating Sequences With Recurrent Neural Networks". In: (Aug. 2013). DOI: 10.48550/ARXIV.1308.0850.

[Hap19] Hannes Max Hapke. *Natural Language Processing in Action. Understanding, analyzing, and generating text with Python*. Ed. by Cole Howard and Hobson Lane. Shelter Island, N.Y.: Manning Publications Co., 2019. 1619 pp. ISBN: 9781638356899.

[Has17] Trevor Hastie. *The Elements of Statistical Learning. Data mining, inference, and prediction*. Ed. by Robert Tibshirani and Jerome H. Friedman. Second edition. Springer Series in Statistics. Description based on publisher supplied metadata and other sources. New York, NY: Springer, 2017. 1745 pp. ISBN: 9780387848587.

[Jam+23] Gareth James et al. *An Introduction to Statistical Learning. With applications in Python*. Springer texts in statistics. Cham, Switzerland: Springer, 2023. 607 pp. ISBN: 9783031387463.

[Jan+20] Valentina Janev et al., eds. *Knowledge graphs and big data processing. LAMBDA Big Data Analytics Summer School*. State-of-the-art survey. The lectures were presented at the LAMBDA Big Data Analytics Summer School (the first edition was held in Belgrade during June 17-19, 2019; the second edition was held online during June 16-17, 2020) - Acknowledgments. Cham: Springer, 2020. 207 pp. ISBN: 9783030531980.

[JM23] Dan Jurafsky and James H. Martin. *Speech and Language Processing (3rd ed. draft)*. 2023. URL: https://web.stanford.edu/~jurafsky/slp3/.

[JR12] Gerhard Jäger and James Rogers. "Formal language theory: refining the Chomsky hierarchy". In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 367.1598 (July 2012), pp. 1956–1970. ISSN: 1471-2970. DOI: 10.1098/rstb.2012.0077.

[KC04] Graham Klyne and Jeremy J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C. 2004. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ (visited on 03/15/2015).

[KST23] Caleb Koch, Carmen Strassle, and Li-Yang Tan. "Properly Learning Decision Trees with Queries Is NP-Hard". In: (July 2023). DOI: 10.48550/ARXIV.2307.04093.

[Kub17] Miroslav Kubat. *An Introduction to Machine Learning*. Springer International Publishing, 2017. ISBN: 9783319639130. DOI: 10.1007/978-3-319-63913-0.

[LAK20] Jurek Leonhardt, Avishek Anand, and Megha Khosla. "Boilerplate Removal using a Neural Sequence Labeling Model". In: (2020). DOI: 10.48550/ARXIV.2004.14294.

[LG90] Douglas Lenat and R. V. Guha. "CYC: A Midterm Report". In: *AI Magazine* 11.3 (Sept. 1990), p. 32. DOI: 10.1609/aimag.v11i3.842. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/842.

[Liu+19] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: (July 2019). DOI: 10.48550/ARXIV.1907.11692.

[LM14] Quoc V. Le and Tomas Mikolov. "Distributed Representations of Sentences and Documents". In: (May 2014). DOI: 10.48550/ARXIV.1405.4053.

[Mak+21] Sasu Makinen et al. "Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help?" In: *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*. IEEE, May 2021. DOI: 10.1109/wain52551.2021.00024.

[MG22] Beatriz M. A. Matsui and Denise H. Goya. "MLOps: a guide to its adoption in the context of responsible AI". In: *Proceedings of the 1st Workshop on Software Engineering for Responsible AI*. ICSE '22. ACM, May 2022. DOI: 10.1145/3526073.3527591.

[MHM18] Leland McInnes, John Healy, and James Melville. "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction". In: (Feb. 2018). DOI: 10.48550/ARXIV.1802.03426.

[Mik+13] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: (Jan. 2013). DOI: 10.48550/ARXIV.1301.3781.

[Mit13] Tom M. Mitchell. *Machine Learning*. [Nachdr.] McGraw-Hill international editions. New York [u.a.]: McGraw-Hill, 2013. 414 pp. ISBN: 0071154671.

[Mot+08] Boris Motik et al. *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax*. Last Call Working Draft. (to be published, may be superseded). W3C, 2008. URL: http://www.w3.org/2007/OWL/draft/owl2-syntax/.

[Ola15] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 03/28/2024).

[Ras16] Sebastian Raschka. *Python Machine Learning. Unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics*. Ed. by Randal S. Olson. Open source community experience distilled. First published: September 2015. Birmingham: Packt Publishing open source, 2016. 425 pp. ISBN: 9781783555130.

[RBP24] Federica Rollo, Giovanni Bonisoli, and Laura Po. "A Comparative Analysis of Word Embeddings Techniques for Italian News Categorization". In: *IEEE Access* 12 (2024), pp. 25536–25552. ISSN: 2169-3536. DOI: 10.1109/access.2024.3367246.

[RG19] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: (Aug. 2019). DOI: 10.48550/ARXIV.1908.10084.

[ŘS10] Radim Řehůřek and Petr Sojka. "Software Framework for Topic Modelling with Large Corpora". English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. http://is.muni.cz/publication/884893/en. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[SS23] Bhuvi Sangam and Savita Sangam. "An Ensemble model to analyze the sentiments of Textual Data for Monitoring and Comprehending Racist Views". In: *2023 6th International Conference on Advances in Science and Technology (ICAST)* (2023), pp. 187–190. URL: https://api.semanticscholar.org/CorpusID:268254694.

[TSL00] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. "A Global Geometric Framework for Nonlinear Dimensionality Reduction". In: *Science* 290.5500 (Dec. 2000), pp. 2319–2323. ISSN: 1095-9203. DOI: 10.1126/science.290.5500.2319.

[Tsu21] Junichi Tsujii. "Natural Language Processing and Computational Linguistics". In: *Computational Linguistics* 47.4 (Dec. 2021), pp. 707–727. ISSN: 0891-2017. DOI: 10.1162/coli_a_00420. URL: https://doi.org/10.1162/coli%5C_a%5C_00420.

[TWW22] Lewis Tunstall, Leandro von Werra, and Thomas Wolf. *Natural language processing with Transformers. Building language applications with Hugging Face*. Ed. by Leandro Werra, Thomas Wolf, and Aurélien Géron. First edition. Beijing: O'Reilly, 2022. 1383 pp. ISBN: 9781098103217.

[UL18] R. Uma and B. Latha. "Noise elimination from web pages for efficacious information retrieval". In: *Cluster Computing* 22.S6 (Mar. 2018), pp. 14583–14602. ISSN: 1573-7543. DOI: 10.1007/s10586-018-2366-x.

[Vas+17] Ashish Vaswani et al. "Attention Is All You Need". In: (June 2017). DOI: 10.48550/ARXIV.1706.03762.

[VGE18] Thijs Vogels, Octavian-Eugen Ganea, and Carsten Eickhoff. "Web2Text: Deep Structured Boilerplate Removal". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 167–179. DOI: 10.1007/978-3-319-76941-7_13.

[Vuk15] Aleksa Vukotic. *Neo4j in Action*. Ed. by Nicki Watt et al. Includes index. - Description based on print version record. Shelter Island, NY: Manning Publications, 2015. 11 pp.

[Wan+20] Yingfan Wang et al. "Understanding How Dimension Reduction Tools Work: An Empirical Approach to Deciphering t-SNE, UMAP, TriMAP, and PaCMAP for Data Visualization". In: (2020). DOI: 10.48550/ARXIV.2012.04456.

[Wu+20] Hejing Wu et al. “Data Analysis and Crawler Application Implementation Based on Python”. In: *2020 International Conference on Computer Network, Electronic and Automation (ICCNEA)*. 2020, pp. 389–393. DOI: 10.1109/ICCNEA50255.2020.00086.

[Yan+19] Yinfei Yang et al. “Multilingual Universal Sentence Encoder for Semantic Retrieval”. In: (July 2019). DOI: 10.48550/ARXIV.1907.04307.

[YC21] Yadavendra and Satish Chand. “Multiclass and Multilabel Classification of Human Cell Components Using Transfer Learning of InceptionV3 Model”. In: *2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*. IEEE, Feb. 2021. DOI: 10.1109/icccis51004.2021.9397165.

[Zen+21] Kaisheng Zeng et al. “A comprehensive survey of entity alignment for knowledge graphs”. In: *AI Open* 2 (2021), pp. 1–13. ISSN: 2666-6510. DOI: https://doi.org/10.1016/j.aiopen.2021.02.002. URL: https://www.sciencedirect.com/science/article/pii/S2666651021000036.

[ZW21] Hao Zhang and Jie Wang. “Boilerplate Detection via Semantic Classification of TextBlocks”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, July 2021. DOI: 10.1109/ijcnn52387.2021.9534308.

Appendix A

Attachments

/
- `Top2Vec.diff` ..... git diff – changes made to Top2Vec
- `scraping_tools` ..... package for scrape processing
  - `sterojos_svp_scraping_tools` ..... package source code
  - `tests` ..... tests for the package
    - `sample_dataset` ..... small artificial dataset for testing
- `extraction`
  - `pipeline_dataset` ..... larger artificial dataset
  - `crawler.py` ..... sample Scrapy spider using the package
  - `example_experiment.py` ..... simple pipeline example
  - `top2vec_experiment.py` ..... Top2Vec pipeline integration
  - `run_top2vec_experiment.py` ..... example of usage
  - `ldia_nn_training.ipynb` ..... training a neural net on LDiA
- `knowledge_base`
  - `tapix-categories.ttl` ..... RDF description of knowledge base
  - `tapix-ontology.ttl`
  - `tapix-tags.ttl`
  - `wikidata-fetcher.py` .. Wikidata client for describing categories
  - `prediction-to-rdf.py` ..... script for adding new merchants
- `prompting` ..... prompting LLMs with TF-IDF filtered dataset
  - `llama-categories-tfidf.ipynb`
  - `mistral-categories-tfidf.ipynb`
  - `category_prompt.txt`

Appendix B

Dataset properties

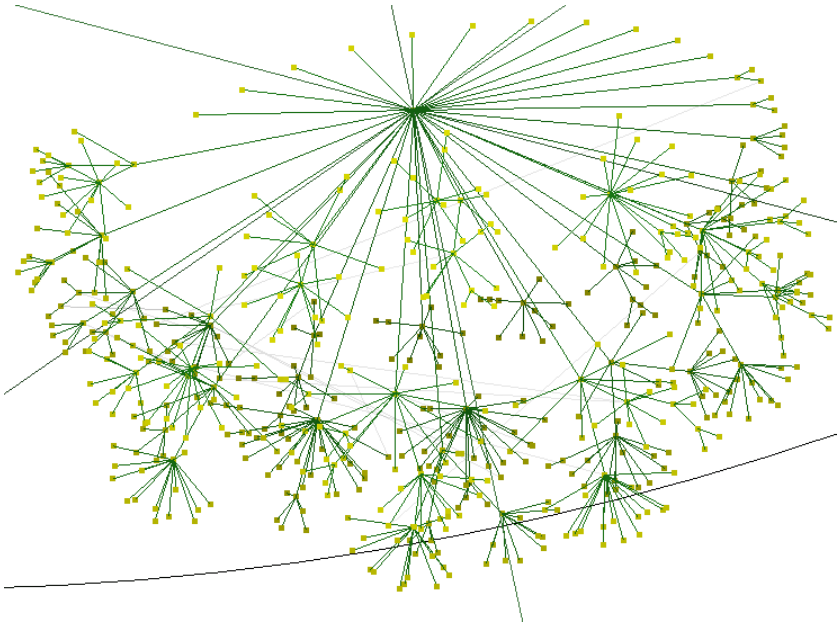


Figure B.1: Visualization of the TapiX category tree made using the Walrus graph visualization tool.

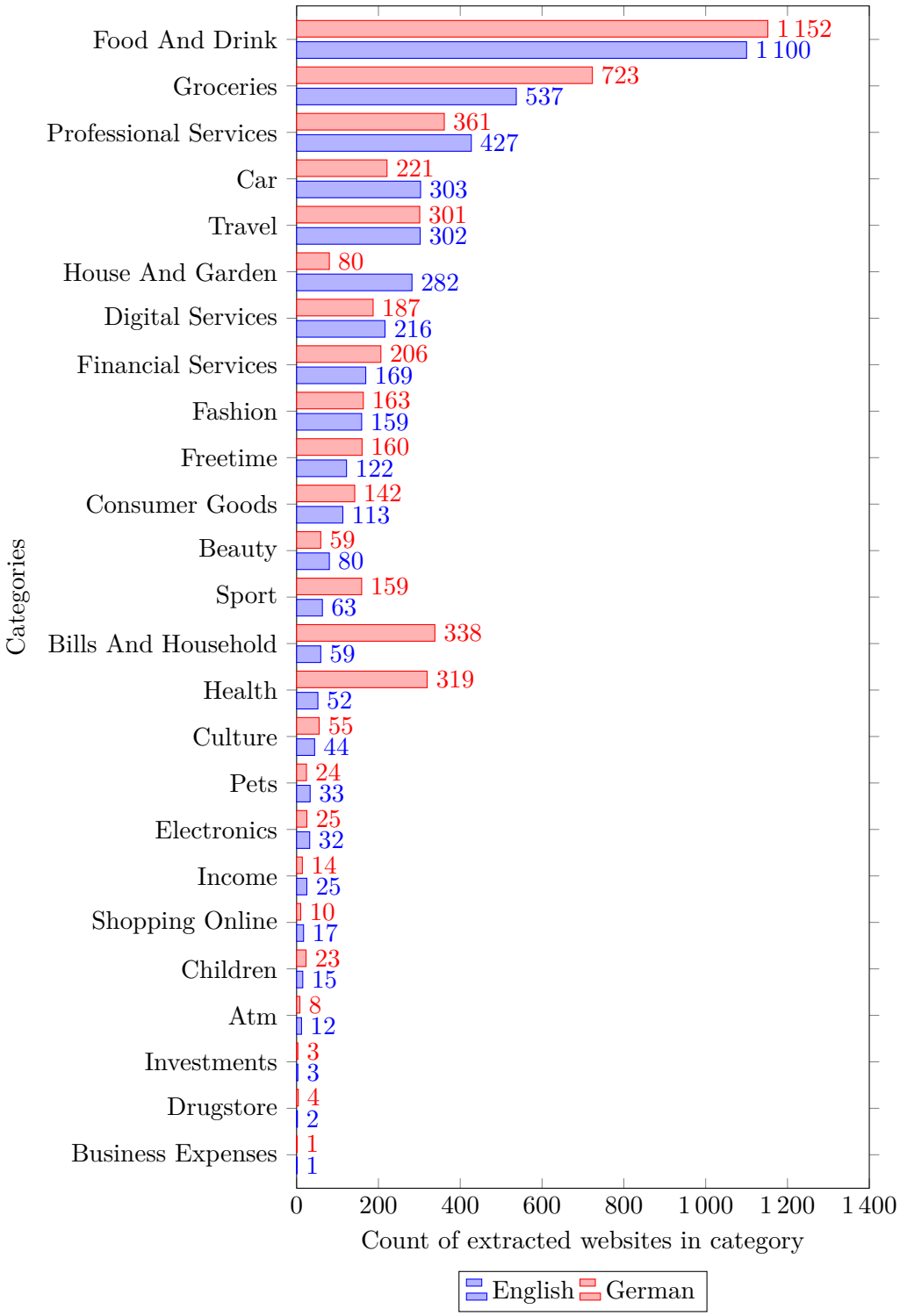


Figure B.2: Distribution of categories in the English-language and German-language datasets.

Appendix C

Document embedding visualizations

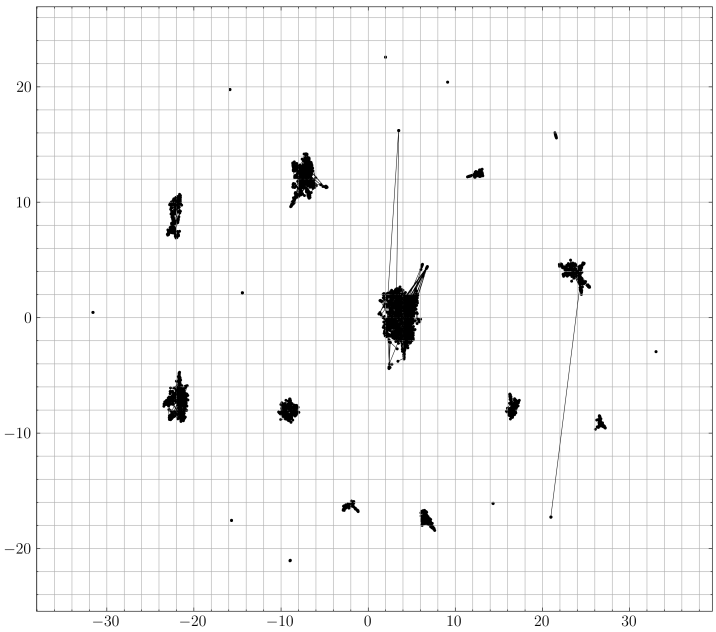


Figure C.1: Visualization of the dimension reduction performed by PaCMAP on an artifically separated dataset. Each point is connected with a line to the point which was its nearest neighbor prior to the dimension reduction.

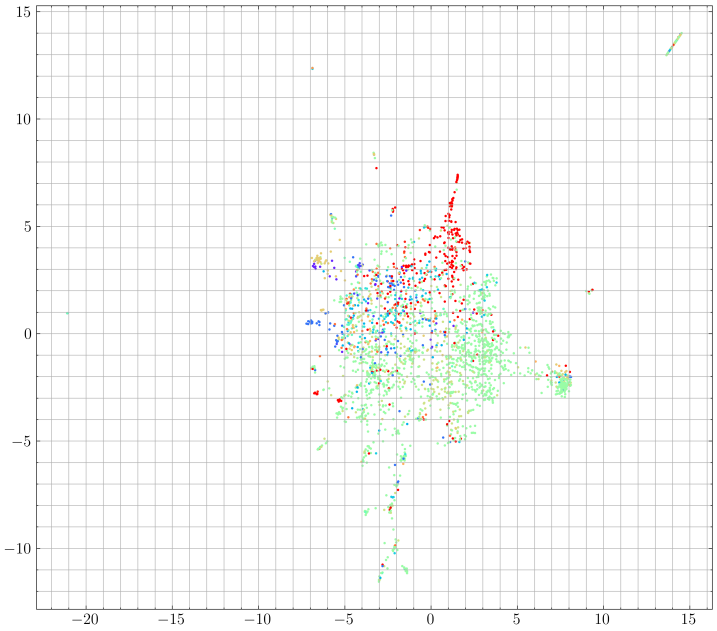


Figure C.2: PaCMAP reduction of the Doc2Vec embeddings of the Spanish-language dataset, extracted using jusText, with points colored by true category.

Figure C.3: PaCMAP projection of Doc2Vec document embedding of the German dataset, extracted using Trafilatura. The points are colored by true category.

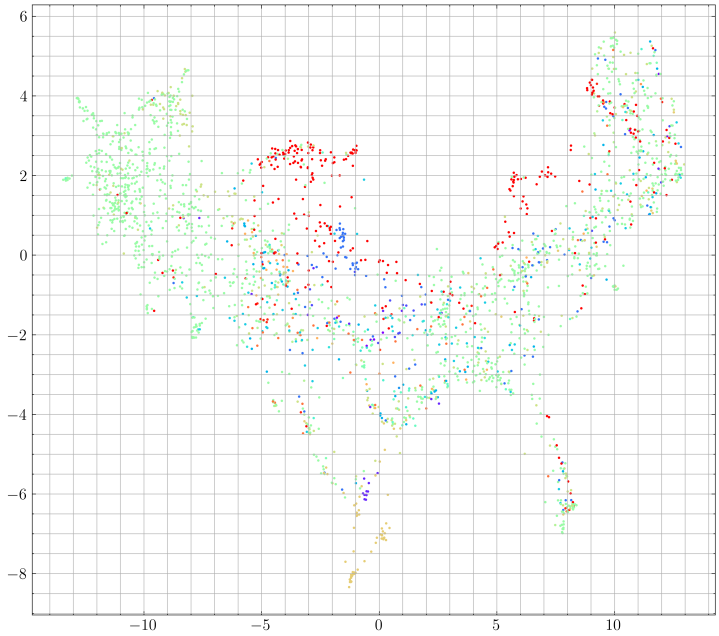


Figure C.4: PaCMAP reduction of the Multilingual Universal Sentence Encoder embeddings of the Spanish-language dataset, extracted using jusText, with points colored by true category.

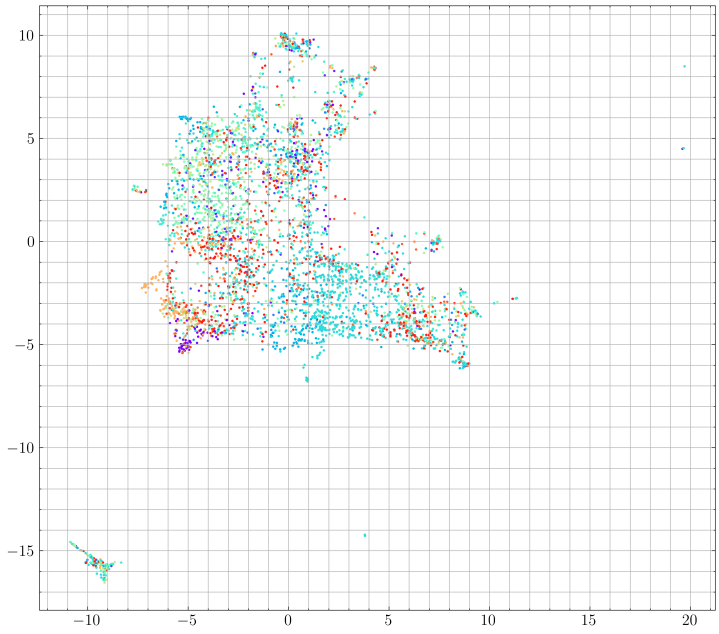


Figure C.5: PaCMAP reduction of the Mistral 7B embeddings of the UK dataset, extracted using jusText, with points colored by true category.

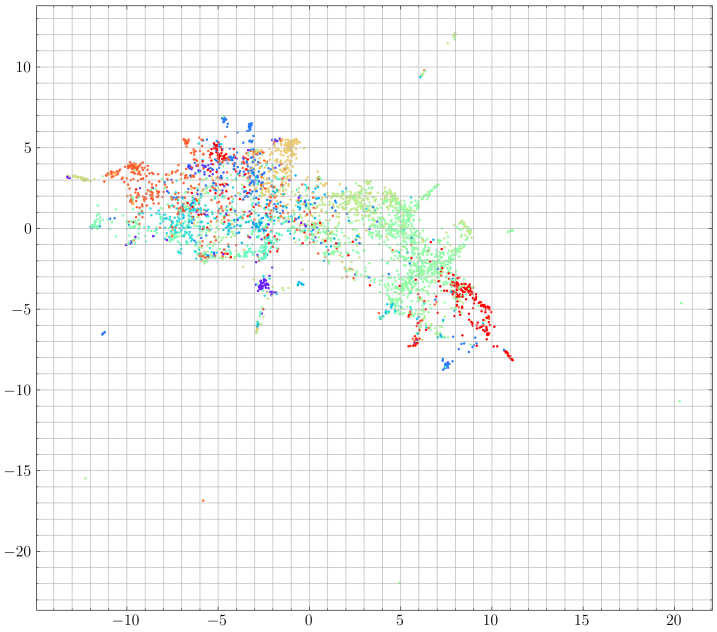


Figure C.6: PaCMAP projection of Doc2Vec document embedding of the UK dataset, extracted using jusText.

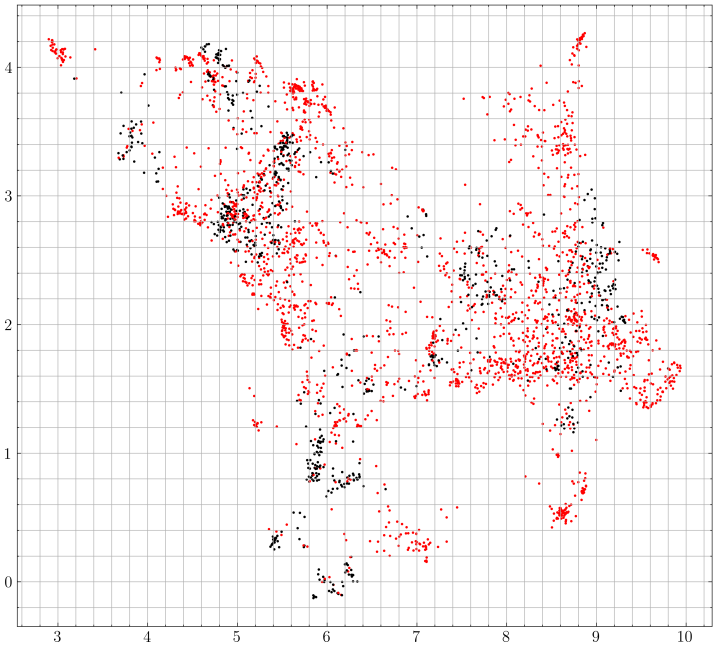


Figure C.7: Matching Top2Vec topics onto human-labelled categories. A document is marked with black if the topic of the document is the best found match for the category of the document, and vice versa. If either of these doesn't apply, the document is marked with red.

Appendix D

SPARQL query for testing the knowledge base

```
prefix txo: <http://www.tapix.io/tapix-ontology/>

select ?category ?tag ?expected_tag_count
where
{
    {
        select
            ?category
            ?total_count
            (count(?merchant) as ?retrieved_count)
        where {
            ?merchant a txo:Merchant;
                txo:hasCategory ?category .
            ?category a txo:Category .
                ?category txo:hasMerchantCount
                    ?total_count .
        }
        group by ?category ?total_count
        order by desc (?retrieved_count)
    }
    bind(?retrieved_count / ?total_count as ?count_ratio)
    ?category txo:hasChild+ ?tag .
    ?tag txo:hasMerchantCount ?tag_count .
    bind(?tag_count * ?count_ratio as ?expected_tag_count)
}
order by desc(?expected_tag_count)
```