Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Extracting Semantic Relations from Wikipedia using Spark

vorgelegt von

Hans Ole Hatzel

MIN-Fakultät

Fachbereich Informatik

Wissenschaftliches Rechnen

Studiengang: Informatik

Matrikelnummer: 6416555

Betreuer: Dr. Julian Kunkel

Erstgutachter: Dr. Julian Kunkel

Zweitgutachter: Prof. Dr. Thomas Ludwig

Contents

1 Introduction

1.1 Motivation

Computationally obtaining knowledge about a words semantics is of interest for automated processing of natural language. Finding synonyms of a given word for example is a problem for which semantic understanding is needed. An automated approach can not trivially relate two words that only have semantic similarity.

Similarly grouping words as belonging to a certain topic seems to require a semantic understanding of the topics domain. As an example: one might try grouping words as relating to certain topics like: imprisonment, geography, or foodstuff. An expert on such a topic knows a set of words associated with it and can apply it accordingly. Finding a group for a given word without a-priori knowledge based on just a dataset of natural language however presents a challenge.

A domain where this concepts can be applied is psychology: tests are used to evaluate humans understandings of semantic and non-semantic relationships between words. A Compound Remote Associate (CRA) task asks the subject to find a common word that builds a compound word with each of the three words. Such a task might look like this: `sleeping/bean/trash` with the answer being `bag`[2] The compound words are not necessarily closely semantically related as can be seen in the above example. The task itself might not require semantic understanding but analyzing the difficulty of such tasks the relationships between the three words are of interest. Questions like "Are tasks with closely related words easier to solve?" could be answered by gaining an understanding of the relation between the words.

Such analysis requires both an approach to gaining semantic understanding as well as the option of generating CRA tasks.

Both these tasks, generating CRA-tasks and extracting the semantic relations of the words involved require a dataset which is ideally representative of natural language. The text of Wikipedia articles is a large corpus on which this analysis should be possible. It might be possible use the language information to infer semantic knowledge about words without a-priori knowledge, the same goes for generating tasks.

This approach comes with many challenges. Large datasets are necessary to allow accurate statements which makes operations computationally expensive. When working with a large dataset like the full Wikipedia text, which is about 12 GB in size, non-trivial computations are slow. Such computations on large datasets need be carried out in parallel to enable quick computations and thereby rapid iteration of algorithms. There is always an issue of data-quality as large datasets usually have inconsistencies, wrong data and missing data. In the case of the Wikipedia, there is the issue of extracting text while excluding the syntax used for styling the text. Other issues one could face with this dataset are spelling and grammatical errors. Different big data solutions are well equipped to deal with these and other related problems.

A popular product from the big data ecosystem is Apache Spark. It is good fit for solving the specific challenges outlined above, since Spark offers rich abstractions to enable rapid development of new work-flows while supporting parallel computation. Spark also comes with machine learning functionality, thereby making the process of analyzing the data much more convenient.

1.2 Goals of this Thesis

This work aims to find CRA tasks and provide data for analyzing the relations of words in them without using a-priori knowledge, purely base on the Wikipedia dataset. This task can roughly be split into the following steps:

1. Preparing the raw data (cleaning and normalization)
2. Determine semantic groups of words
   (a) Build lists of distances to surrounding words

   (b) cluster words by their surrounding words

3. Extracting compound words for automatic creation and solving of the CRA-task
   (a) Recognize nouns
   (b) Check for each word built from other nouns how often it exists in the corpus
4. Finally, based on this exploration, constructing efficient alternative pipelines for Spark to solve these tasks
5. Apply all procedures to both German and English datasets

1.3 Outline

In Section 2 of this work, background information that is needed for the understanding of the whole work is discussed. This discussion includes the technology and techniques that were used as well as some basic concepts and terms. Section 3 gives a short overview off other academic work that is related to this thesis.

Section 4 proposes a design for achieving the goals of the thesis. This design consists of two main pipelines one for extracting compound words and one for extracting semantic relations.

Section 5 goes on to discuss the concrete implementation of the proposed design detailing issues and interesting aspects of the implementation.

In Section 6 finally, the implementation's speed and the quality results are evaluated. Section 7 wraps the work up by providing a summary of previous sections and giving a final evaluation of the results in the conclusion.

2 Background

This section provides background information on the technology, techniques and datasets used in this thesis. Section 2.1 discusses the Wikipedia dataset that was used. Section 2.2 explains the technique of part-of-speech tagging. Section 2.3 through Section 2.5 focus on different miscellaneous concepts needed for the tasks at hand. Section 2.6 gives an introduction to the data mining techniques that are used in this work.

2.1 Wikipedia

Wikipedia is an online encyclopedia by the Wikimedia Foundation. It is available in a variety of different languages, and contains a large variety of articles on different topics. Wikipedia articles are created collaboratively by the community. [18]

The Wikimedia Foundation also offers free data dumps of Wikipedia content on a dedicated website[17]. This way it is not necessary to scrape the data from the web pages for analysis, thereby reducing strain on the servers as well as simplifying work for anyone analyzing the data.

Among the formats offered are full SQL database dumps but also XML files with the articles' contents. The XML files are well suited for most analysis as they contain the articles text as well as some metadata. The XML files are also relatively easy to parse.

The text contained in the dumps is not free of the syntax elements used for controlling the visuals and behavior of the article. Some examples of these elements include but are not limited to: italic text, bold text, tables, images and links to other pages. There are existing third party solutions for both extracting the article text from XML dumps and performing the data cleaning required to work with the raw text.

2.2 Spark

Apache Spark is a data processing engine for parallel execution in cluster environments. Spark relies heavily on an abstraction called RDD (resilient distributed dataset). These are an abstraction over a set of data which is shared among multiple machines enabling parallel operations that are transparent to the programmer.

The data model of RDDs is to represent data as a sequence of key-value pairs. Each element of an RDD is a pair of a key and a value.

For parallel execution, Spark relies on one driver program that handles the sequential sections of a given program and schedules execution of parallel sections via tasks. As Figure 1 shows, this is achieved by using a *SparkContext* that schedules executors on worker nodes. During execution the *SparkContext* sends workloads in the form of tasks to executors.

For scheduling, the Spark driver needs a cluster manager to communicate with the workers. When configuring a Spark environment, the user can choose which cluster manager to use. One of these is the YARN resource manager, there is also a built in one available in Spark that works without external dependencies.

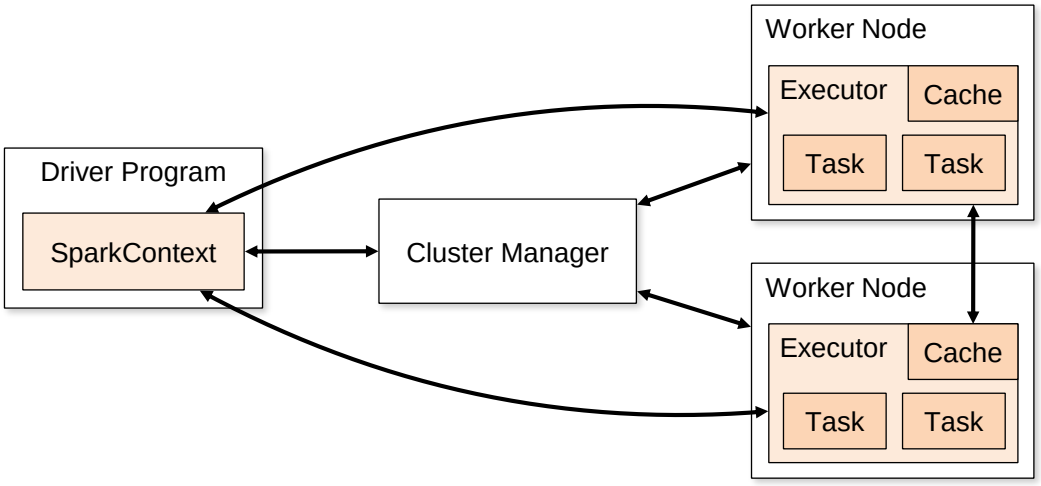


Figure 1: Structure of Spark [3]

There are two basic types of operations on RDDs. The first one is transformations, these are lazy operations on RDDs returning a new RDDs. Lazy evaluation is generally used to describe computations that are only performed when the result is needed. In this case, it means transformations are not applied right away, but instead the operations are only applied when the content of an RDD is actually needed. [3]

The second type of operation is an action. An action would, for example, be needed to print the results of a computation. Actual evaluation of a transformations on an RDD only takes place when an action is performed. Only actions actually require computation on the data. All transformations are just stored, and given the original data and the history of transformations the resulting data can always be computed whenever it is needed for an action.

As a history of all desired transformations is always available it is possible for Spark to optimize their execution. The sequence of operations can be represented as a DAG (directed acyclic graph) it is not just a sequence of transformations as there are operations that take two RDDs as input.

```
>>> data = [(i, i / 2) for i in range(1, 1000)]
>>> my_rdd = sc.parallelize(data)
>>> new_rdd = my_rdd.map(lambda x: (x[0], x[1] * x[1]))
>>> new_rdd.take(5)
[(1, 0.25), (2, 1.0), (3, 2.25), (4, 4.0), (5, 6.25)]
```

Listing 1: A very simple demonstration of the Spark interactive shell

Listing 1 shows a simple Spark program as executed from the interactive shell. First the user creates a list of pairs with the first element of each pair being a number from 1 to 1000 and the second element being half of the first number. This data is then parallelized, meaning an RDD with it is created where the keys are the first elements of the pairs. After that each value of in the RDD is squared. The keys meanwhile remain untouched. Finally, the take action is performed returning the first 5 results pairs. The map operation will - given a Spark environment configured for this - execute in parallel on multiple machines.

Table 1: Typical Spark Transformations and Actions [11]

Transformation	Description
`map`	Map over all tuples in the RDD
`flatMap`	Map a function over all pairs with the option for each iteration to return multiple resulting pairs
`groupBy`	Pair of a key and a list of all values with this key

Action	Description
`reduce`	Aggregate all elements using a reduction function.
`collect`	Returns all elements of an RDD.
`count`	Count elements of an RDD.
`take`	Retrieves a specific number of elements from an RDD.

While this is a very simple computation, there is still one very interesting aspect to it: the map operation won't actually be computed for all data, instead only the pairs actually needed for the take operation will be computed. Due to the lazy evaluation that is used, only these five pairs will be computed and only as soon as the take operation is performed.

This property can be very useful when working with small subsections of datasets in longer computations it is, however, often lost by performing operations like sorting which require all tuples to be known. One such operation that will lead to the need to evaluate all operations is the `groupBy` operation (listed in Table 1).

RDDs in Spark use a concept called data lineage to ensure failure resilience. Each dataset can be recomputed at any time as both the origin of all data and all subsequent operations on this data are known. This means if any worker node should fail at any time during execution the data that is lost can just be recomputed. Spark thereby has built in failure safety for both software and hardware faults. The smallest unit of data that can be recomputed in this manner is called a partition. Partitions are a subset of key value pairs which are guaranteed to be in the same task, and are thereby executed on the same worker node and in the same executor.

Thanks to its interactive shell Spark is well suited for both interactive data exploration on large datasets and subsequent machine learning analysis. [19]

Spark itself is written in Scala but provides programming interfaces for using Spark in R, Java, Python and Scala.

2.2.1 PySpark

To use Spark in a Python program, Spark's Python interface PySpark is required. PySpark is essentially built on top of the normal Spark infrastructure. Multiple Python process are launched on each worker, one for each RDD. The driver launches a `SparkContext` which is responsible for communication with other nodes using Py4J[**pysparkinternals**].

Py4J is an interface to call Java functions from Python. Communication between different nodes is still done using the original Scala functions which are called from Python using the Py4J Java-Python bridge.

For communication Python objects are serialized using the `cloudpickle` library. This library unlike Pythons standard library's `pickle` allows serialization of more advanced structures such as lambdas. Lambdas specifically are a requirement for Spark as they are used to supply Spark operations (e.g. `map`) with functions to apply.

2.3 Linguistics

Throughout working with the large dataset of natural language the need for terms to describe specific linguistic features will arise.

2.3.1 Proper Nouns & Common Nouns

Common nouns are nouns that are used to refer to a class of objects, or a single object of a class (house, books, water etc.). Proper nouns however are nouns that are specific to one individual object, meaning they can be thought of as names. Actual names of people (Mary, Hans, etc.) as well as those of places, companies (Hamburg, Yahoo, etc.) and the like are considered proper nouns[12].

2.3.2 Compound Words

Compound words are words that are combined from two words. This combination is not limited to nouns but usually involves at least one noun. Such a word might be the combination of the words 'home' and 'cooking' to form 'home cooking'.

In the English language, a compound word is usually separated by spaces. Some words can also be written with a dash or in one word, usually for very commonly used words.

In German, the involved words are combined into one without the use of spaces.

2.4 CRA

CRA tasks are a task developed by psychologists to asses the thought process in problem solving. A test subject is given three words and is asked to find a forth one that builds a compound word with all of the three. An example of this could be `fox/man/peep` with the answer being `hole`. The goal is to observe what is called an insight solution. Insight solutions as opposed to non-insight solutions involve the person not actively coming up with an answer by working through possible solutions but instead having a sudden "Aha! Experience"[2].

2.5 Part-of-Speech Tagging

While the taggers are only used and no understanding of the internals is necessary a small overview should nonetheless improve understanding. Part-of-speech tagging aims to be able to assign words tags, which indicate the words grammatical function in a sentence. A tagger should be able to distinguish between nouns, verbs, adjectives, prepositions, etc.

Tagging a sentence like "`Where did I put the milk?`" should return something like:

```
[('Where', 'WRB'), ('did', 'VBD'), ('I', 'PRP'),
    ('put', 'VBD'), ('the', 'DT'), ('milk', 'NN'), ('?', '.')]
```

The only tag that will be interesting for this work is the "NN" tag that indicate singular, common nouns.

A very simple tagger might just use tags of known words from an already tagged training set. This approach does however not work well since some words might appear in different functions depending on the context and will also not work for any words that were not present in the training data. More advanced models take the surrounding words into account (using Hidden Markov Models).

The tagger Stanford tagger, which will be used in the implementation, is based on a work by Toutanova et al.[16]. It takes additional measures to improve accuracy for unknown words and reaches an accuracy of around 86.76% for unknown words. The overall accuracy meanwhile reaches of 96.76% for almost all tags.

2.6 Data Mining

2.6.1 K-Means Clustering

Generally the aim of clustering is to split a set of data points into different clusters of similar data. K-Means is an algorithm to perform clustering. It assigns each of the data points to one of K-clusters. The algorithm takes an integer value k and a set of data points as input and returns a class for each data point.

A method to find out which value for k is well suited for the data is the elbow method. It is a visual method that considers the mean squared error (error being the total distance of all points to their respective cluster center) and evaluates it's change with increasing values for k.

2.7 TF-IDF

TF-IDF, the term frequency-inverse document frequency is a value to indicate the significance of a word in a document[9].

$$IDF_i = log_2\left(\frac{N}{n_i}\right)$$

$$TF_ij = \frac{f_ij}{max_k f_{kj}}$$

The inverse document frequency IDF for a term is calculated by dividing the total number of documents by the number of documents N the term i appears in. The term frequency is calculated by diving the number of occurrences of i in the document j by the maximum frequency of any term in j. The TF-IDF for a given document can then be calculated by multiplying the two terms.

3 Related Work

3.1 Explicit Semantic Analysis

The paper "Computing Semantic Relatedness using Wikipedia-based Explicit Semantic Analysis" by Gabrilovich and Markovitch describes a method to gain semantic knowledge about words and concepts based on the Wikipedia corpus[4].

The technique called Explicit Semantic Analysis (ESA) aims to be able to find out which concepts are related to a given input text. After filtering out some short or insignificant articles the remaining articles are equated to concepts. The idea is to now be able to find for any given word the concepts that most closely relate to it. The word "equipment" for example will turn up "Tool" followed by "Digital Equipment Corporation", "Military technology" and "Camping" as closely related words.

For each concept a vector is build which indicates which words occur in it. In this vector each word has a weight (its TF-IDF) assigned to it indicating how closely related it is. A reverse index is built up from this to quickly know for any word which concept it is related to.

Based on this data the concept any given text has can be related to concepts. This relation can be calculated by adding up the measure of relation of all word (weighted by their TF-IDF) in the article to all known concepts. The concept with the largest sum is assumed to be the one which is most closely related to the text.

In this way, the system was able to classify the following texts: *"U.S. intelligence cannot say conclusively that Saddam Hussein has weapons of mass destruction, an information gap that is complicating White House efforts to build support for an attack on Saddam's Iraqi regime. The CIA has advised top administration officials to assume that Iraq has some weapons of mass destruction. But the agency has not given President Bush a "smoking gun," according to U.S. intelligence and administration officials."*[4]

The concepts it came up with as most closely related to the text were:

1. Iraq disarmament
2. Yellowcake forgery
3. Senate Report of Pre-war Intelligence on Iraq

3.2 Word2Vec

Word2Vec is a project that aims to gain understanding of words by representing them as vectors in an effort to help with all kinds of natural language processing tasks. It is based on the paper "Efficient estimation of word representations in vector space"[10]. Generally each word is characterized by a vector of the words typically surrounding it. These vectors are generated by training a model using a natural language corpus. The resulting vectors have useful properties that are related to both the syntactics and the semantics of the words they describe. For example, subtracting the vector of the word "France" from the vector of "Paris" and the adding the vector of "Italy" will result in a vector that is close to the one of "Rome".

Before training the model a vocabulary needs to be set this can be done by taking the most frequent terms in a corpus.

When training the model, the surrounding context of every word w in the vocabulary is explored. A window size describes how many of the surrounding words are taken into account when training, even words within this window may have lower weights depending on how far they are away from w. Word2Vec does this by implementing two basic approaches, the Continuous Bag-of-Words (CBOW) and the Skip-gram model.

CBOW The Continuous Bag-of-Words trains a neural network to predict the current word given the context (i.e. the surrounding words).

Skip-gram The Skip-gram model trains a neural network to predict the surrounding words given a word w.

Both approaches are implemented using single hidden layer neural networks.

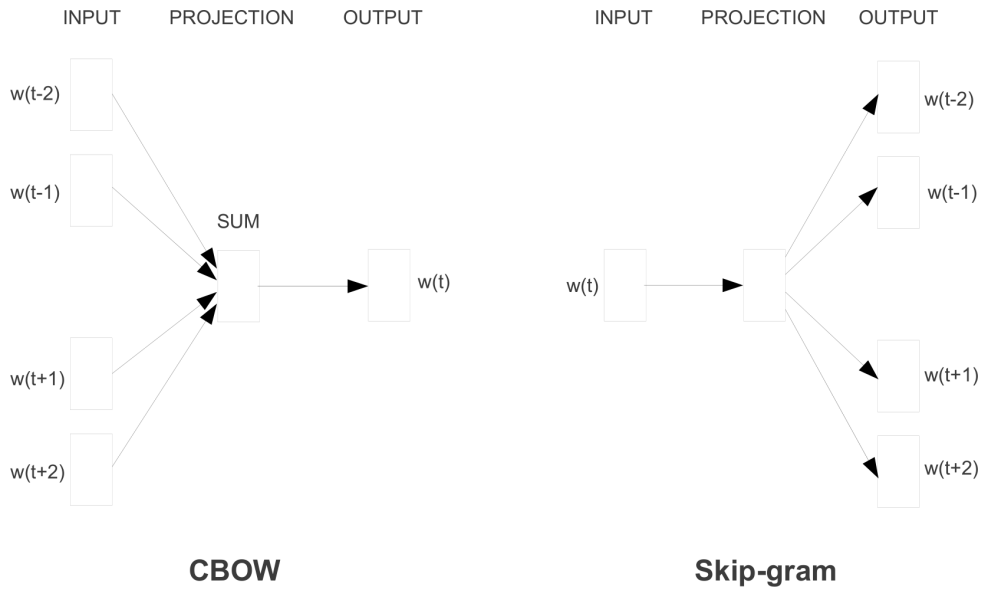


Figure 2: Continous Bag-of-Words and Skip-gram model compared [10]

When glossing over a lot of details in the actual implementation of the networks you can think of the two models as being the reverse of each other. In the Skip-gram model, the output layer of the neural network given the input word w gives the probability of each word in the vocabulary being in the context of w. In the Continuous Bag-of-Words model, however, the input consists of the surrounding words. These probabilities can be understood as a vector describing the word.

Words that are similar will have similar vectors and can accordingly be clustered using clustering algorithms. As alluded to in the introductory sentences of this subsection, these vectors can even be used in algebraic operations the researches were able find that the word "smallest" is related to "small" in the same way that "biggest" is related to big. The only operation that was necessary for this was this algebraic operation (where $vector(w)$ is the vector of likely surrounding words for w): $Vector(\text{"}biggest\text{"}) - Vector(\text{"}big\text{"}) + Vector(\text{"}small\text{"})$

This results in a vector, and when looking at the total vector space for all words the closest vector was $Vector(\text{"}smallest\text{"})$.

Calculating these vectors over large a large corpus comes with many challenges. A variety of optimizations were done to this model to enable training on large data sets in reasonable time. Many small enhancements are also developed to improve the quality of the model.

4 Design

This section is dedicated to the design of the parallel and sequential programs developed for this work. It does not focus on the actual implementation which will be discussed in Section 5
Section 4.1 will discuss the perpetration of the input data, Section 4.2 then lays out the process of extracting compounds from the prepared data.
In Section 4.3, the creation of CRAs from the compound dataset is discussed.

4.1 Preparing Raw Data

Before processing the data it has to be prepared in some kind of readable format that is suitable as input for a Spark application.

Authors of Wikipedia articles use a special syntax to specify the articles appearance, this syntax is known as the wiki markup. To analyze the Wikipedia text, all syntactic elements of the wiki markup need to be removed. As there are many different elements in this syntax, it is a sizable task to remove all elements correctly while not removing the actual text.

For this reason, all data for this work was preprocessed using a third party solution called Wikiextractor[1]. This tool is a Python script which was modified for the purpose of this work to output a CSV-file with one element for each article.

CSV was chosen as a file format for its simplicity and since CSV files allow for convenient importing of the data into processing environments like Spark.

There is, however, one problem when working with full Wikipedia articles inside a CSV files. The existence of newlines inside the articles text. Spark has no ability to handle newlines inside CSV-values. Despite there being the theoretical - and even standardized option - of ignoring newlines inside of CSV fields [13]. An actual implementation carries problems in Spark's distributed context.

The issues with implementing newline compatible CSV reading in Spark is mostly a performance issue as splitting tasks across jobs is non trivial when for each newline a check has to be performed as to whether or not it is inside a quoted field.

To avoid these problems, newlines are removed in this initial step before actual processing with Spark begins. While there is some loss of information associated with this, the proposed analysis does not heavily rely on newlines. The results should thus not be noticeably affected.

The initial step of preparing the raw data is performed sequentially without the help of Spark. As this step is only performed once and finishes within a reasonable time, therefore sequential processing is acceptable. The result of this first step and also the input for pipeline described in the next section, is a CSV file of articles with names and the full article text. The article text has all wiki markup removed.

```
12,https://en.wikipedia.org/wiki?curid=12,Anarchism,"Anarchism is a
    political philosophy that advocates self-governed societies based
    on voluntary institutions. These are often described as stateless
    societies, although several authors have defined them more
    specifically as institutions based on non-hierarchical free
    associations."
```

Figure 3: Shortened version of the article on anarchism as stored in the CSV file.

Listing 4.1 shows the structure of the CSV file, note that the text is quoted to allow commas inside the field. Any line breaks in the articles where replaced with ten spaces each. The example is split across several lines merely to fit on this documents page. In the actual CSV file all newlines were removed.

4.2 Pipeline for Finding Compound Words

A pipeline for finding compound words will be built as a single PySpark application. This program will aim to find a list of compound words C_i for a base word N. Said base word could be any given noun. The result should therefore be a list of Compounds for any given noun N: $(N, [C_1, ...C_n])$

For our overall design, it is assumed that a compound word consists of at least one noun. No check will be performed on the second words function or validity.

Each compound word is assumed to be made from one noun and a prefix or suffix word. The two words might be additionally be connected by an epenthesis with a length of up to three.

Further, it is also assumed that a noun is at least three characters in length, this seems a reasonable limit to not exclude common words like "man" or "tie". In fact no word in Bowden & Jung-Beeman's list is shorter than three characters. Landman's list only features two words with the length two, the German words 'Ei' (*egg*), and 'Öl' (*oil*)[8][2]. This means that some impact on the results can be expected, there is however a large upside in terms of processing time and dismissing potential false-positives.

One additional issue is that in English compounds as used in CRAs can be built up from several nouns separated by a space as well, meaning that an additional way to filter out all nouns that occur next to each other is needed. If those words are found, they could just be added to the list of words as a single string with both words separated by a space, the methods that will be discussed in this section all still work in the same basic way.

Figure 4.2 represents the whole pipeline, the following subsections will each discuss one of the steps in the pipeline.

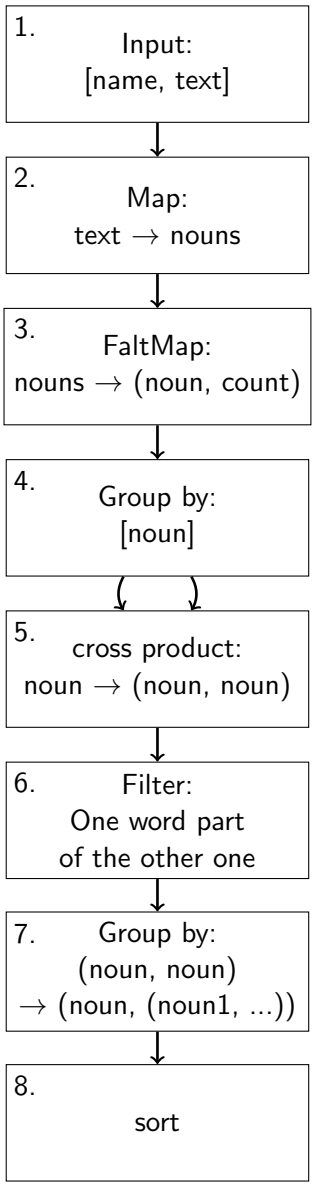


Figure 4: Pipe Diagram

4.2.1 Identifying Nouns

Step 2 in Figure 4.2 shows the noun identification step.

The identification of nouns will be performed in a single map operation in Spark. Spark takes care of performing this operation time efficiently by making use of all available hardware using parallel operations.

A single map operation is used to convert each articles text into just a list of nouns that occur in the article. All nouns are passed on while all other words are discarded.

For this operation it is necessary to discern for each word whether or not it is a noun. Two basic approaches to this identification can be identified:

1. Capitalized words are nouns (for German texts)
2. Part-of-speech tagging

The former method works by relying on the fact that in German language nouns are capitalized. Therefore a heuristic can be formulated that classifies each word starting with a capital letter as a noun.

This simple heuristic has a lot of potential issues in the form of edge cases. The simplest case of these exceptions is a word at the start of a sentence, which while it may or may not be a noun, will always be capitalized.

This specific exception can easily be handled by not taking words at the beginning of sentences into account. Single false-negatives are assumed to be of little importance, as it is likely that in a sufficiently large dataset each word will occur multiple times and, barring systematic errors,

subsequent occurrences of the same word should lead to them still being included in the data. False-positives however will have a direct negative impact one the quality of the resulting data.

The latter approach, part-of-speech tagging, makes use of more sophisticated models for determining the grammatical function of words. In the case of the Stanford part-of-speech tagger, which is used exclusively in this work, it is possible to use a pre-trained model to tag both German and English texts. The Stanford tagger additionally supports a variety of other languages.

The output of the Stanford part-of-speech tagger is a sentence where each word has one tag from a predefined list of tags associated with it. The set of tags for German texts for example contains the relevant tags listed below [14].

```
NN      normales Nomen  Tisch, Herr, [das] Reisen
NE      Eigennamen      Hans, Hamburg, HSV
```

A decision has to be made on whether or not proper nouns are included in the noun extraction process. For this work it was decided to include proper nouns, as CRAs tend not to contain any proper nouns.

While it certainly is feasible to construct compounds using very established brand names these are edge cases and show up in neither Landman's [8] nor Bowden & Jung-Beemann's paper[2]. In the interest of staying close to established CRA tasks proper nouns are therefore discarded in this step. Alongside proper nouns the pipeline will also discard all plural forms to avoid any duplicates where both the singular and the plural form are included.

The output when tagging a sentence looks as follows:

```
[(Und', 'KON'), ('so', 'ADV'), ('schriebb', 'VVFIN'),
('ich', 'PPER'), ('eine', 'ART'), ('Geschaichte', 'NN'),
('namens', 'APPR'), ('Und', 'KON'), ('täglich', 'ADJD'),
('grüßt', 'VVFIN'), ('das', 'ART'), ('Murmdsdtier', 'NN')]
```

Note that even though the words contain very uncommon spelling mistakes the tagging can still work. Even for unknown words the tagging can work as the tagger also relies on the context of a word.

A combination of the tagging and the heuristic approaches is also possible, in fact a part-of-speech tagger might make use of the capitalization in its classification model. The significance of false-positives as opposed to false-negatives, as discussed above, means that when combining the two approaches the sensible option is to only keep a noun if both approaches classify it as such. Using the part-of-speech tagging obviously is the more computationally expensive method. In fact the tagger that will be used makes use of the capitalization heuristic the improve it's performance on unknown words [16].

Whether or not using part-of-speech tagging will, in the grand scheme, entails significant runtime penalties for the whole pipeline will be subject to further analysis in Section 6.

For English texts, the program will have to solely rely on the output of the part-of-speech tagger as no simple capitalization heuristic is possible.

In some sense the tagger model constitutes a-priori knowledge of the language but it is required to find out which words are nouns, especially for English texts. There is no way around using this piece of a-priori information.

4.2.2 Counting Words

Step 3 and 4 in Figure 4.2 are involved in counting words. The input data for this step was already converted to a list of nouns for each article, as the step before removed all non-nouns. This list of the form `["apple", "pear", "apple"]` for each article now needs to be converted into a global list of noun occurrences across all articles. This means that globally the RDD has this form:

```
("fruits article", ["apple", "pear", "apple"])
("food article", ["meat", "meat", "apple"])
```

Following the same example the article from above would result in the following list `[("apple", 3), ("meat", 2), ("pear", 1)]`. To generate a set of tuples from each article the `flatMap` operation will be used. This means that each of the tuples from the list will now be a tuple in the RDD. The usage of words per articles is not of interest, instead the counts need to be combined across all articles. This `flatMap` operation is visualized as step 3 in Figure 4.2.

The count for each word, meaning the number of global occurrences, will be carried through all further calculations. For simplicity these counts won't be listed in this document unless they are essential for the discussed operations.

In the implementation these counts will consistently be carried through the whole computation pipeline.

Step 4 then removes duplicates in the RDD which now contains word counts. This transformation from lists of words to a global list of each word with its count can be implemented in two seemingly reasonable and distinct ways.

- The first one is the creation of dictionaries with word counts for each article and combining these using a reduce operation. This means that the result initially is an RDD of dictionaries of the form $\{noun_1 : k_1, noun_2 : k_2, ...noun_n : k_n\}$ where k_i is the number of occurrences of $noun_i$ in a given article.
- The second option, by contrast, proposes the creation of pairs of the form $(noun, n)$ for each word. The final result is then attained using Spark's `groupByKey` transformation and then counting the resulting values in a single map operation.

The second option is the one visualized in the pipe diagram.

In the evaluation section, these two approaches will be compared based solely on their performance, as both yield the exact same results and the implementations can therefore be used interchangeably. The second option is presumed to be the faster one as the `groupByKey` operation should have the potential to better leverage the distributed nature of Spark.

4.2.3 Forming Pairs of Words

Since the last section resulted in a list of nouns with no duplicates a cross product across this RDD of all nouns will yields an RDD with tuples of nouns. Each noun forms a pair with each other noun. For each such pair of words a check of whether or not the first word is a sub word of the second one is performed.

```
("apple", "banana")          is discarded

("apple", "appleseed")       is passed on
("apple", "applesauce")      is passed on
```

An additional check of whether the resulting word is actually a noun could improve accuracy but would come at the cost of performance and would also discard compounds made from say a noun and an adjective (e.g. "sweetpotato"). The naive implementation of this would in fact require another cross product, an operation which is expensive and also rises quadratic in complexity as the number of nouns rises.

If one base word b is either prefix or suffix of the other word c, then c is assumed to be a compound word formed from b and another word. If a tuple forms a compound in such a way it, is kept in a resulting RDD for further computation.

This method even gives the application the ability to find compounds that involve non-nouns despite them being filtered out in the previous step. While the other (non-noun) word is not explicitly recognized and stored both the compound and the noun are. This means that words like "sweet-potato" or the German word "Grünkohl" (*kale*) which consist of a noun ("potatoe" and "kohl") and an adjective ("sweet", "grün") will not be discarded.

4.2.4 Alternative for Forming Pairs

As the previously described step involves a cross product across all known words the quadratic complexity should, given a large enough dataset, represent the bottleneck in this pipeline. For this reason, while the naive implementation is useful for thinking about the whole pipeline, a solution with the potential to perform better should be considered.

To improve performance it would be beneficial to compare each word with as few words as possible. There is one feature than can easily be used to narrow down the number of words before performing the cross product: the words length. Under the assumption that nouns are at least 3 characters in length each word can only be part of compound that is at least 3 characters longer then itself.

To use this property, for a faster yet equivalent process all words are first grouped by their length. For each length a cross product with all sufficiently larger lengths can now be created. This is done for every length in the dataset such that there is an RDD with the results for each length. A union is formed across all these RDDs.

The result is equivalent and the computation might be faster. Detailed performance analysis is performed in Section 6

4.2.5 Grouping

Compound words are grouped such that for each base word b all compounds that were in distinct tuples before are aggregated in one list. This step can simply make use of Spark's built in `groupBy` transformation.

Adhering to the example above this results in the following data:

```
("apple", ["appleseed", "applesauce"])
```

4.2.6 Sorting

To facilitate manual review and to make finding important words easier, a final step is performed to sort the results. This does not actually impact the results but should make manual review easier and enable the evaluation of the pipeline's results.

First the compound words for each base word are sorted in descending order by their frequency, thereby showing words that occur more frequently at the beginning of the compound word list each resulting tuple. This is done locally for each line and can therefor be achieved with a simple map operation.

It is also helpful to sort all lines by some measure of the base word's importance or frequency. There are two proposed measures to do this:

1. Sorting by the number of occurrences of the base word.
2. Sorting by the total number of occurrences of all compound words associated with a base word.

This second sorting operation is performed globally and can be done with Sparks `sort` operation. Ultimately these operatics are only important for human analysis of the results.

A final output tuple after this last computation step might look like this:

```
("potato", ["sweetpotato", "potatosoup"])
```

Or when including the counts which were not discussed for most of the pipeline.

```
(("potato", 200), [("sweetpotato", 80), ("potatosoup", 60)])
```

Note that the compound word counts don't end up to the base word count, as occurrences of the base word are also counted outside of compounds.

The whole compound pipeline generates an output file with tuples of base words and their compounds. The results of this pipeline can be saved and different calculations can then use the results. Making it possible to iterate on the programs without having to execute the entire pipeline again.

4.3 Generating CRA Tasks

The initial goal will be to generate the CRAs that are easiest to solve for each base word. This process will work with the data generated by the compound pipeline.

4.3.1 Psychological considerations

The CRA tasks are used in psychology to assess the decision making progress. They are specifically focused on producing so called insight solutions where the participant gets the answer from a subconscious thought process.

CRA's can have different difficulties, some can be solved by more then 90% of participants in just 15 seconds. Others however are solved by less then ten percent of participants, even after half a minute.

For humans there are two aspects that influence a tasks difficulty.

1. How well known the words are to the subject
2. Whether or not the task is homogeneous or heterogeneous. In a homogeneous puzzle all compound words solution is *either* prefix or suffix, not both. In heterogeneous tasks on the other hand the solution shows up as both suffix and prefix across the three challenge words.

One additional aspect that could be linked to the difficulty is how well the three words are connected in the human mind. If all three words have to do with whether events the subject might find it more easy to search for common prefixes and suffixes.

It is assumed for the purpose of the extraction that the difficulty is just based on the knowledge of the words. The (in)homogeneity of the compounds is easy to check afterwards, and the aspect of relation between the words will be discussed in Section 4.5. For we assume that a task that is as easy as possible should be generated.

4.3.2 Extracting CRA Tasks

When ignoring whether the solution is homogeneous or not, the easiest such task should define itself by the three words that are most recognizably linked with the base word. There are two potential ways to achieve this.

1. Take the three most commonly occurring compounds for each base word
2. Take into account the global rarity of other (non-base) word involved in a compound

The first approach is perhaps the simplest solution, it assumes that the three base words are most commonly used in conjunction with the base word are the most recognizable words.

Using the sorted results from the compound pipeline the first approach is as simple as picking the three leftmost words in the list.

Given this list of the form: $((w, 500), [(wa, 100), (bw, 90), (cw, 50)], (wd, 1)])$ where w is the base word, the three words for the task would be a, b and c.

We define their score to be

$$count(x)/count(w)$$

where x is the count of the specific compound and w the count of the base word. Yielding us, for now, the exact three highest scoring words as described above.

While the assumption of the most common words being the most recognizable is reasonable it does not take into account that common compounds might be common in multiple contexts and therefore bad at identifying a specific word.

For example, the prefix "grand" can be used in many contexts, "mother", "father", "son", "daughter", "canyon". When a tasks solution is one of the above words "grand" might not work well in narrowing the possible solutions down. Thereby making the task harder to solve correctly. In conjunction with two other words, it might still be easy to come up with the correct solution, but with the goal being a CRA task that is as easy as possible to solve each word should be good at identifying a unique solution.

Potentially these first three words are also commonly used in other words and thereby might not good at unambiguously identifying the base word. The second option aims to counter this by taking into account how often these words in other compound words. The proposed metric for this is normalizing the occurrence of each word by how often it appears in other compounds. Expressed more concisely the local score for a word is the number of it's local occurrences divided by it's number of global occurrences. $score(w) = \frac{count(w)_{local}}{count(w)_{global}}$

This second approach is expected to result in triples of words that are better at uniquely identifying a solution. There is however the issue that words that occur very infrequently can have a large impact on the result. If there is some completely unique word that just occurs once and only in conjunction with a single base word the word will be in the resulting triple.

Consider again the example from above:

$$((w, 500), [(wa, 100), (bw, 90), (cw, 50)], (wd, 1)])$$

Assume that d does not occur anywhere outside of the compound wd and only occurs once in the compound, the relevance score would be $\frac{count(d)_{local}}{count(d)_{global}} = \frac{1}{1} = 1$, making it the highest possible value (as the local occurrence can never be more than the global one). A single occurrence of some rare word should not influence the resulting tuple that much. This does also make the process susceptible to errors as a single word could always be a parsing mistake or even an error in the input data. Existing CRA tasks tend to focus on relatively common words as the subjects need to know the words.

Some combination of the two approaches seems useful. We propose to multiply the local score by the global score resulting in an overall score for each word. Seeing as both scores are relative to a total count and always less then one a this operation results in the desired behaviors for globally rare and locally common words being considered together.

In this way the compounds can receive new scores which indicate better how well suited they are for an easy CRA task. To generate a CRA one just has to take the three top scoring compounds.

There are however possible ambiguities in the solutions to these tasks. While the proposed scoring system works against this by favoring uncommon words that happen to be common in this specific compound it does by no means guarantee unambiguous solutions.

These ambiguities fall in two categories.

1. Two base words come up with the same three best scoring words.
2. One base word comes up with its top scoring three words, but these words actually have a larger total score in some other base word.

The first issue can be resolved with simple pruning, the second problem however requires the use of a different approach.

4.3.3 Pruning of results

While the combination of both approaches approach should yield lists that are better suited for identifying a base word based on the three returned words there is no guarantee for uniqueness. Two compounds might still generate the same three words, such a situation is not desirable as whoever is given the task has no single, clearly correct, solution.

In other words: providing a one-to-one relationship from the three words to the base word should be the goal as otherwise the tasks have no clear single solution. It should be a goal to create tasks that allow only one obvious solution.

How can this be achieved? The proposed solution is to perform one final pruning step on the result. This pruning is done in a simple, iterative and greedy algorithm.

1. Check for ambiguities
2. For each set of ambiguities: Replace the highest scoring word in each CRA with the next highest scoring one (for this base word) not yet in the task.

Step one means checking the whole dataset for ambiguous initial words. This can be done relatively efficiently using `groupByKey` operations. Two base words with the same three top ranked words represent an ambiguity in this context.

- For each of the collected set of ambiguities the total score of the used words is calculated
- For the list with the lower overall score the last word in it is replaced by it's successor.

For each of the collected ambiguities the second step is performed, the words scores in both base words lists are compared. The only way that an ambiguity can not be resolved in this way

This process is performed iteratively up to a set limit of iterations given the long lists it seems unlikely that an ambiguity can not be resolved. There is however no guarantee for ever achieving complete uniqueness. The algorithm also has the potential for a very long runtime when two lists are identical. Safeguards might have to be implemented to protect against a very large number of iterations. If this methods yields no good results it can be assumed that for the purpose of this analysis the words are essentially the same (as they can not be distinguished by the words they appear in compound with).

4.3.4 An Alternative Approach

The approach described above is very simple and probably yield decent results. There are however cases were the pipeline could generate tasks with a answer that is not actually the most fitting one. The problem is that each base word is considered individually, and while the proposed algorithm is good at coming up with reasonable CRAs for any given base word it is not checked whether or not three words fit well with another base word.

This situation could arise if one base word for example has very evenly distributed compound counts, making the three highest scoring compounds relatively low scoring. The other base

Let us construct an example to demonstrate this:
$((w_1, 800), [(w_1a, 20), (w_1b, 10), (w_1c, 5), (w_1d, 4), ...])$ This example generates an overall score of: $\frac{20}{800} \cdot \frac{20}{100} + \frac{10}{800} \cdot \frac{10}{100} + \frac{5}{800} \cdot \frac{5}{50} = 0.006875$ where the first term in each summand is the local score and the second one the global score (which for this example was chosen at will).

The other word w_2 could, while having three different highest scoring words, still have the higher score for a, b and c. Take this example:
$((w_2, 800), [(w_2z, 400), ..., (w_2a, 40), (w_2b, 10), (w_2c, 5), (w_2d, 4), ...])$. It is obvious that this produces a larger total score when looking for a base word to the words a, b and c. $\frac{40}{800} \cdot \frac{20}{100} + \frac{10}{800} \cdot \frac{10}{100} + \frac{5}{800} \cdot \frac{5}{50} = 0.011875$

The obvious way to solve this is by checking the proposed tuple against all other compound words and assigning it to another base word should the tuple score higher for that base word. The approach changes from finding a triple of words for a base word to finding a base word for a triple of words.

The approach we propose to do this works similarly to this naive approach. For each base word and using the proposed scoring system a high scoring subset of compound words is taken. Each possible three item combination of these is checked against the whole dataset. For each of these three tuples there is one base word which is the highest scoring, this is the task that is generated. This approach has the advantage that there is a list at hand of how close in score the related words are thereby allowing words where solutions are not unambiguous enough to be discarded.

The disadvantage is that it might not produce one CRA for each base word and the worse performance.

This approach does truly guarantee that within the existing dataset any CRA has a unique mapping from three words to the answer.

4.4 Finding solutions for CRA Tasks

Based on the output of the original compound pipeline there is for every base word a list of compounds with scores. These can be used to solve both real world as well as the auto generated tasks.

4.4.1 Generated Tasks

Assuming all tasks are generated with the algorithm proposed in Section 4.3.4 and with the same dataset that is available for solving, it should always be possible to unambiguously find the correct solution. Even for the first approach in Section 4.3 this would be expected to work reasonably well.

For real world tasks, generated by something else then the exact algorithm above this approach might however fail. It might not always be possible to find the correct solution due to differences in sampled data or assumptions made during the task's creation. For tasks not created with the exact same data or with another algorithm the goal is for the solutions to still be reasonable and ideally correct.

4.4.2 Real World Tasks

The ability to solve real world CRA tasks relies on the assumption that the dataset in use is a good representation of the language that is actively used.

In theory, the approach is simple the algorithm operates on the result of the compound pipeline. Given the three challenge words the algorithm works through the whole dataset of compounds and their base words. For each base word it checks how many of the compounds in the list are formed with one of the challenge words. If none are the base word can immediately be discarded as a possible solution.

If two or three words match the same scoring system as proposed in Section 4.3 is used.

There is however some concern in choosing the sum of the three individual scores as a metric. This means, that words that are very rare will have little to no impact on the result. One option we will implement to avoid this is by preferring any base word with three matches over all those with just two matches, no matter the individual scores.

An additional way to counter this effect is by using an asymptotic function to scale the results. We use the base 10 logarithm, using any such function results in very large values having less and less an effect, meaning that even small scores can make an impact. A base word with three compounds that all have scores in a medium range might now win out over one where two of the words barely occur while one is very common.

This measure in effect works towards favoring balanced tasks where compound is somewhat easy to come up with over those were one is easy but the rest is hard. To ensure that this algorithm still

works for solving the generated tasks this same modification is also applied to the scoring used in Section 4.3.4.

In Section 6.3, the algorithm will be tested on published lists of CRAs.

4.5 Finding Semantic Groups in List of Words

To help assess the difficulty of a compound word it would be useful to have an understanding of how closely the words in a puzzle are related to each other. The assumption being that three words that are close in meaning make it easier to solve the compound puzzle.

To illustrate this Figure 5 provides a simplified view of how one might group compounds of the word game. The first image is a rough map of semantic closeness of each word to each other mapped into two dimensions. You can see how some of the words clearly are related: "minigames" a subset of videogames, "gameboards" and "gamemasters" are both related to games played in a social setting. In the end, one can try to group these terms into groups where all words in a group are closely related. The right hand side of Figure 5 is an example of one possible way these could be grouped.

While Figure 5 was created by hand as an example the aim would be to create something simmilar in an automated fashion. Knowing how the terms are related might help in assessing the difficulty of CRA tasks.

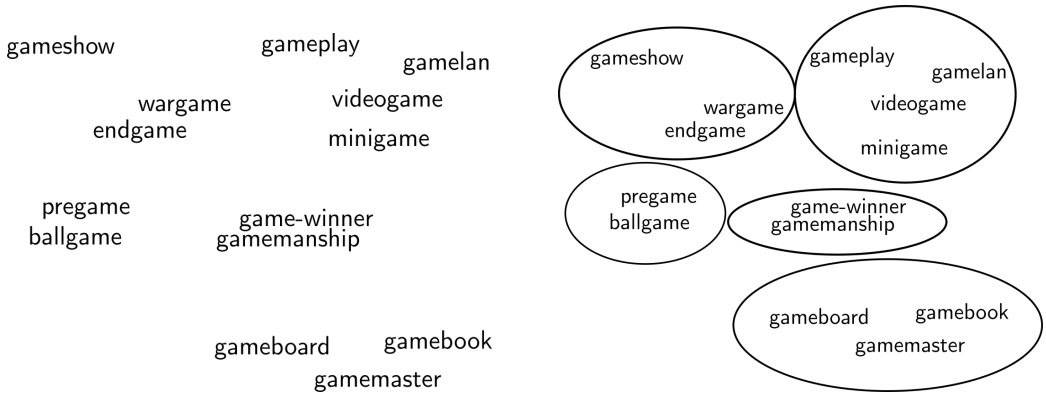


Figure 5: Example for semantic groups in compounds with the base word "game".

The problem of finding semantic groups inside a list of compounds can be generalized to finding semantic links in in a list of words. One approach to mine the semantic characteristics of a word is to look at the words it occurs with. This is could be done by looking at which words occur in the same document with a given word. Wikipedia articles however can span multiple topics, consider for example the article "Germany", it spans multiple topics from culture to politics and history. The fact that two words occur in the same article is probably not a good indicator that they are in related in terms that relate to similar things. The vocabulary is the list of words we want to cluster and are looking for related words to. For each word in the vocabulary, we consider a window of context-words surrounding it.

This approach could be considered a very simplified version of the same approach as used by word2vec, so it will be interesting to see if it can still yield some results.

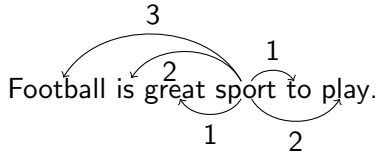


Figure 6: Scores based on proximity in a text.

The exact metric that will be used is distance to the word in question, if the word "team" occurs in a text close to the word "tournament" this means they are probably related. A short distance

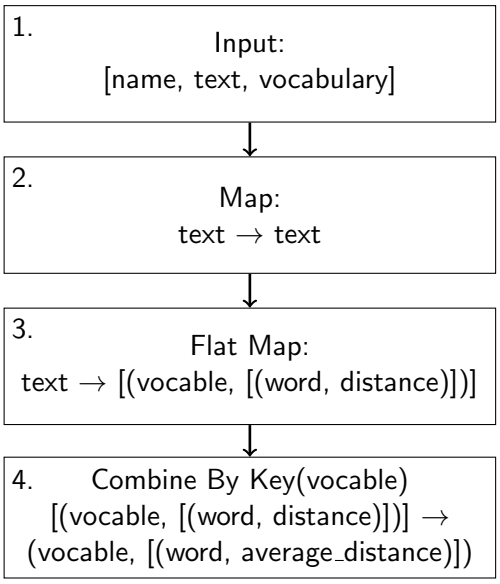
signifies an assumed close semantic relation. Using this approach each word w can be characterized by a list related of words $r_1, ..., r_i$ and their average distance to the word w. Obviously, not every occurrence of w will have all related words in close proximity. In this case, a large default value can be assumed. A reasonable value seems to be double of the maximum distance at which words are considered to be in close proximity. The implementation will look at words that are up to 15 words away in either direction, meaning a given word has up to 30 neighbors. The result is a list of words with associated average distances for each related word $(r_1, dist_1), ..., (r_i, dist_i)$ across all occurrences of w.

This vector of distances can be used as a set of features on which to perform clustering, in this manner any list of words can be separated into semantic groups. Using the K-Means algorithm this means that given an integer value for k the words should be sorted in k meaningful clusters.

Choosing a good value for k for the general case of all list of words is not possible. Ideally k should be the number of groups that the words would naturally be sorted into by a human.

The approach general approach has been discussed. The concrete pipeline design in PySpark (see Figure 7) will look as follows:

Figure 7: Pipe Diagram



1. Loading of input. Each tuple consist of the text and the name of the article. Additionally there is also a list of vocabulary provided, while this is not strictly part of the tuple (instead it is retained locally in each worker) it is still listed here for simplicity. This list is provided by the user, these are the only words which surroundings will be examined. The output will accordingly be a vector for each word.
2. The text is preprocessed. Potentially filtering is applied (details in Section 4.5.1).
3. Each document is transformed into a set of words from the vocabulary. Each vocabulary has a list of words found in it's surroundings with the corresponding distance. This list can be thought of as the vector.
4. The vocable's entries generated from each article are merged vocable. The lists of word distances (i.e. the distance vectors) are merged. The merge operation entails averaging the distance, taking a large default value if it doesn't appear in one of the vectors.

4.5.1 Filtering to Improve Word Vectors

A potential problem is bias towards common words making the resulting data less meaningful and clustering more error prone. Common words occurring in close proximity may or may not actually characterize the semantics of a word. Therefore clustering should be performed with a few added filtering operations.

Two approaches to improve results are outlined:

- Remove words based on part-of-speech tagging
- Counterbalance word distances with a measure of their frequency (TF-IDF score)

Both approaches are relatively simple.

In the former approach, words are excluded from analysis either based on a their tag in part of speech tagging. For example any prepositions might be deemed non indicative of the words semantics.

The latter approach, proposes using a measure of word frequency to counter balance a word being closer on average. The goal is to weigh the distance with some measure of how common a given word is. As common words are expected to show up closer to w they have less significance and less weight is given to them.

A common measure for doing this is TF-IDF score[9]. While the described approach doesn't include a notion of document frequency a distance can still be weighted with the inter document frequency using this formula:

$$score = \frac{averagedist(w)}{idf(w)}$$

It is similar to the TF-IDF measurement in that it weights the measure of importance local importance (here the average distance instead of the count) by the global importance. As the inverse document frequency grows the term is more rare, diving by this term results in rarer words being counted as the closer words, compared to very common words.

Summary: *In this section, a pipeline for finding compound words that are in use in the Wikipedia was designed. Methods for using this data to both generate and solve CRA tasks outlined. Finally a second pipeline was designed that aims to find out about a words semantics based on the words it usually occurs in conjunction with in the Wikipedia corpus. The implementation of all this will be discussed in Section 5. The resulting program's quality will be evaluated in Section 6*

5 Implementation

This section focuses on the implementation of the design discussed in Section 4. Some specifics that were deemed interesting are discussed in detail. In Section 5.1, the deployment of the software is discussed, Section 5.2 discusses the process of implementing PySpark applications and leads into Section 5.3, a section devoted to the testing setup that was created to speed up development. Section 5.4 will discuss specific issues that were encountered with part-of-speech tagging implementations. Finally, Section 5.5 will discuss which versions of the software involved were used and the consequences of these choices.

5.1 Deployment

The whole project with all pipelines is submitted to YARN using a GNU-Make makefile. The makefile packages external dependencies like the Stanford part-of-speech tagger into archives and submits them along with all Python files. Different pipelines are realized as different make targets whereas options are supplied to the specific pipeline using an environment variable (in this case: `OPTIONS`). This design means that all pipelines can be completely separate, as no shared startup code is needed.

5.2 Working with PySpark

Debugging of Spark applications is time consuming due to a few issues:

1. Long startup times of the Spark environment
2. Potentially long execution times when using non-minimal input data
3. Checking the program's output is non-trivial (when using YARN) as the log files contain a lot of irrelevant information and need to be retrieved with a yarn-command.

For PySpark specifically, the additional issue of little to no static correctness checks makes bugs harder to catch before execution. This made the development process tedious as every single bug, be it just a misspelled variable name, can take a while to find out about and identify in the source code.

Most of these issues can, to some degree, be diminished by not submitting the application in cluster mode but instead running it in a local Spark environment. That way the logs are printed directly to standard out instead of being collected by YARN. The overall turnaround time is a lot lower as the logs can be checked more easily and the submission takes less time.

The same issues, however, still take effect just to a lower extend. For this reason, a testing setup is still a very useful tool to catch errors early in the development process and thereby avoid debugging altogether.

The following subsection will discuss the testing setup in detail.

5.3 Testing

To alleviate the problem of long turnarounds in debugging large portions of the Python code are covered by tests. The testing toolkit that was used is PyTest. While large portions of the code can be covered by simple unit tests without the requirement to interact with Spark, some sections of the code require interaction with Spark.

To test these sections involving Spark, two main approaches were used:

- Mocking very simple parts of the Spark API
- Using Spark's local mode for testing

The first approach results in very fast tests but is less reliable as details of the Spark API's behavior could invalidate the test results with mocked operations. This also requires the developer to actually write a mocked version of a part of the Spark API. Writing custom mocked implementations can potentially represent an error source.

The second method, while requiring no further code, leads to the tests' execution being much slower. A Spark context has to be launched which comes with significant overhead in the order of multiple seconds. Having fast tests is very convenient as it enables the developer to quickly check the correctness of new code. While it is much slower the second method does more accurately represents Sparks behavior and help catch more bugs that rely on the interaction with Spark.

To combine the advantages of both approaches testing in the code base is done either in a fast mode which runs all those tests which don't require Spark, meaning both simple unit tests and those that use a mocked version of the Spark API, or alternatively in a second mode. The second mode executes all the tests including those that require Spark. This results in Spark environments being launched in local mode.

Passing as Spark context to certain tests is implemented using test fixtures. Fixtures are a construct that is part of PyTest and were designed for the use case of preparing an environment for unit tests. With the fixture being defined, any test function that takes the argument "sc" automatically gets passed an already initialized Spark context.

The implementation of this fixture is shown in Listing 5.3. Essentially, the function just creates, sets up and returns a `SparkContext` object.

```
8  @pytest.fixture(scope="session")
9  def sc(request):
10     if pytest.config.getoption("--no-spark"):
11         pytest.skip("Skipping Spark tests (--no-spark specified)")
12     conf = SparkConf()
13     sc = SparkContext(conf=conf)
14     request.addfinalizer(lambda: sc.stop())
15     return sc
```

Listing 2: Fixture to supply tests with Spark contexts

There are a few additional important details, the `pytest.skip()` call skips all those tests involving a Spark context whenever the option `--no-spark` is supplied. This enables running tests in two modes as described above, enabling even quicker turn around for a subsection of the tests.

There is also a `finalizer` that stops the Spark context after all tests were executed and thereby ensuring termination.

The fixture is set up using a function annotation which also takes an additional argument, the scope. The scope specifies how often the "sc" function will be called to create a new `SparkContext`. If it is set to "session" this means the function will only be executed once per testing session, which means all tests taking the "sc" argument will share one `SparkContext` instance. [7]

```
3  def test_wikicsv(sc):
4      wikipedia = ["""42,https://some.wiki.url,Article Name,"Some "Text"\
5                   in here, no newlines though\"""] * 3
6      csv_rdd = sc.parallelize(wikipedia)
7      w = wiki.read_csv(csv_rdd)
8      assert w.take(1)[0]["text"] == "Some \"Text\" in here, no newlines though"
```

Listing 3: A test can use the fixture simply by taking `sc` as an argument.

Listing 5.3 shows how simple a test can be created using this setup. The function just takes "sc" as an argument and is handed an initialized `SparkContext` during execution. It then uses this context to created an RDD with some sample data which is passed to a function that is being tested.

5.4 Part-Of-Speech Tagging

The Python module `NLTK` offers a rich set of functions for processing natural language. Among them functions for performing part-of-speech tagging in some languages. The tagger that supports German is a Python API for the Stanford part-of-speech tagger.

As it turns out however, the NLTK module for this specific tagger simply opens a Java subprocess for call to the tagging function.

```
# Run the tagger and get the output
stanpos_output, _stderr = java(cmd, classpath=self._stanford_jar,
                                                    stdout=PIPE, stderr=PIPE)
```

```
# Call java via a subprocess
p = subprocess.Popen(cmd, stdin=stdin, stdout=stdout, stderr=stderr)
```

Listing 4: A subprocess is called whenever tagging a new string using NLTK's Stanford tagger module

Listing 5.4 shows two excerpts from the NLTK source code, the first one shows the `java` function being called to run the tagger. The second excerpt is from said `java` function and launches subprocesses.

This approach, while certainly being relatively easy to implement, has the second huge advantage of not needing any extra dependencies to communicate with the Java process. All communication is performed using functionality of the Python standard library, which offers a simple API for calling subprocesses and communication with them. The approach does, however, also result in very slow processing speeds as each time the `tag` function is called to tag a text a subprocess is launched.

There are three reasons this has large performance impacts:

1. Process creation overheads
2. JVM initialization overheads
3. Repeated calls to the function

While the first reason applies for any process on all major operating systems the second one only applies to Java processes which do have some additional overhead. The third reason is specific to this application, as there are millions of articles and the tag operation is performed individually for each article the performance overhead as caused by the first and second reason is more signification because the code is executed multiple million times.

There are several approaches to reduce the performance impact:

1. Merge many articles to end up with fewer documents that are individually tagged, resulting in fewer subprocess calls. This does however represent some level of complexity in implementation especially when considering that the number of individual documents must not be too low in order to still make full use of the performance benefits of parallel execution.
2. An alternative to this strategy that is more broadly applicable is using another measure to avoid launching individual subprocesses. Instead, one can also use any interprocess communication method to keep using the same process for tagging multiple artilces. One method for this would be to would be manually implementing some type of serialization in Python and deserializing this data on the Java side. This process should work but involves writing a fair amount of custom code.

3. A simpler method, still using interprocess communication, instead relies on a library to perform the communication between processes. The library that was used is Py4J, it enables communication between a Python and a Java process, thereby making it possible to call Java methods from Python with little effort on the programmers side. After some initial setup Java methods can simply be called from Python code and behave almost exactly like Python functions.

An initial test was done to access the viability of the Py4J approach over just using a subprocess for each call.

Calling the same code using the Py4J bridge a speedup of about 30 was achieved in a small scale test as seen in Section 5.4. For this test, a subprocess with a small Java to perform the tagging was launched from Python code. The same tagging operation was then performed using Py4J.

This test was performed on an laptop using a test file with 4578 words in 90 lines, with one call of the tagging function for each line. Compare this to the results when tagging the whole test dataset in a single call and it becomes apparent that the startup time is indeed the bottle neck for the tagging. To properly asses the actual performance impact of this one must compare the size of documents in the test dataset, which are just a single sentence per each, to those of the complete dataset. No matter how long the documents are, one can save significant time by not starting up a process with a JVM instance for each individual call to the tagging function. Consider tagging 6

Table 2: Trivial test of a Py4J based implementation compared to the NLTK one with subprocesses. A grand total of 4578 words is tagged, either in one call or split up into 70 calls.

Method	Number of Calls	Total time in seconds
Py4J	70	1.043
Subprocess	70	36.427
Py4J	1	1.023
Subprocess	1	1.893

Million articles where the startup time is 1 second this ads up to overall over 1,500 hours of CPU time.

These considerations mean that the NLTK implementation would entail significant performance impacts and we will therefore not use it.

After deciding to not go with the NLTK based solution there is still another option then using Py4J for subprocess communication. Instead, a tagger that is written in either C or Python could be used, as either of those could easily be called from the Python code.

While this tagger in C or Python could potentially have better performance there appears to be no finished tagger for German texts available in either language. Training a tagger (and even more so building the software for it) is however outside of the scope of this work.

5.4.1 Wrapping a Java Based Tagger

The Py4J approach allows usage of the Stanford part-of-speech tagging solution. It does however still come with potentially large processing overheads as communication between the Java and Python process is slow. This approach also makes for a strange infrastructure, as each of the Python processes launched by Spark the program launches an additional Java processes purely to handle tagging.

Integration with the existing Spark infrastructure turned out to be a non trivial problem. It is not easily possible to import another version of Py4J inside a PySpark environment. PySpark itself makes use of Py4J to integrate with the Spark implementation which is largely written in Scala. As the Java side also requires a library for communication which come with any installed version the first step was to install the most recent version via `pip`, the Python package manager. This version came with a jar-file containing the library to handle the Java side of the communication. As the PySpark environment does, however, not enable the import of other versions of Py4J and the versions where not matching there were communication errors.

It seems like a good idea to just use the current Py4J version for application internal communication and import it in Python using a different name. In principle, this is as easy using `import py4j as some_name` to import a disjunct version of Py4J. As however the PySpark version takes greater precedence when importing, one has to adjust the Python path to prefer the version installed by pip.

Changing the `PYTHONPATH` variable however does not affect which location the Py4J module is imported from in PySpark. In the standard Python environment, this works as expected and enables you to choose the priority which path to import modules from. PySpark however seems to ignore this variable, perffering built in modules. This was probably done to avoid users importing another version of the same library under the same name, thereby braking communication with the Java side (which would then be running a potentially incompatible version).

As a result the only remaining option for importing is importing from a specific global path. This is possible using the `importlib.util` module in Python 3.

Manually importing does however not work in this case as Py4J itself relies on Py4J being in the Python path (it imports modules of itself using absolute names). This could be worked around by patching the Py4J source code, which is a processes that involves quite a bit of work, you can also no longer rely simply installing the most recent version but instead will have to patch every new version.

The only method that ended up solving this issue was using the built in Py4J version of PySpark. No jar file could be found, in the local Spark 1 installation. As Spark 2 however did ship with this jar it was possible to use this. This did however involve hard coding of class paths in the implementation as otherwise the correct Py4J jar would not be found.

To make interacting with the Tagger as easy as possible a wrapper class was written that both launches a Java subprocess and handles all communication with it, providing ordinary Python methods for tagging.

As this is one of the more important pieces of the infrastructure the code will be explained briefly.

In the `__init__()` function initializes a Tagger instance and starts a Java subprocess using the `launch_java()` function.

The main method of interest however is the `tag_string()` method, it calls a Java function using the Py4J bridge and does some processing on the output to return it as a Python data-structure.

```
2  import py4j
3  import py4j.java_gateway
4
5  class Tagger():
6      """
7      A tagger to tag Strings using the Stanford Java tagging backend.
8      Be aware: Each instance of Tagger will create it's own Java process.
9      Don't create excessive amounts of Taggers as this will hurt performance.
10     """
11
12     def __init__(self, model_path):
13         print("NEW TAGGER with %s" % model_path)
14         self.launch_java()
15         py4j.java_gateway.java_import(
16             self.gateway.jvm,
17             "edu.stanford.nlp.tagger.maxent.MaxentTagger"
18         )
19         self.java_tagger = self.gateway.jvm.MaxentTagger(
20             model_path or "models/german-fast.tagger"
21         )
22
23     def launch_java(self):
24         port = py4j.java_gateway.launch_gateway(
25             classpath="""/usr/hdp/2.5.0.0-1245/spark2/jars/py4j-0.10.1.jar\
26 :stanford/stanford-postagger.jar:stanford::""",
27             die_on_exit=True,
28             jarpath="/usr/hdp/2.5.0.0-1245/spark2/jars/py4j-0.10.1.jar"
29         )
30         self.gateway = py4j.java_gateway.JavaGateway(
31             gateway_parameters=py4j.java_gateway.GatewayParameters(port=port)
32         )
33         return self.gateway
34
35     def tag_string(self, text):
36         """
37         Tag a string, this method will also split the string and return a list tuples.
38         Each tuple is a word and it corresponding tag.
39         """
40         # The last character in the tagged string, for some reason, is always a space
41         # It can just be stripped using rstrip (This is safe in case this bug is fixed in the future)
42         if len(text) > 0:
43             tagged = self.java_tagger.tagString(text).rstrip()
44             res = []
45             for wt in tagged.split(" "):
46                 pair = wt.split("_", 1)
47                 if len(pair) == 2:
48                     res.append(pair)
49             return res
50         else:
51             return []
52
53     def __del__(self):
54         self.gateway.shutdown()
```

Listing 5: The tagger wrapper class allows simple interaction with the Stanford tagger

5.5 Software Versions

This subsection discusses which versions of the software involved were used and why.

5.5.1 Spark

Due to the Spark 1 version not shipping with a Py4J jar and version mismatch issues that come with this the program ended up only working with Spark 2. Spark 2 is supposed to bring large performance

improvements and is therefore the more sensible choice either way. While Spark 2 brought a lot of changes, the RDD API did for the most part not change. It is expected that given the Py4J issue can be solved the pipeline would also work with Spark 1.

5.5.2 Python

Regardless of the Spark version, there is the option to choose the Python version to be used. While many software projects still rely on Python 2 there is no real reason for most new applications to use Python 2 (unless any desired libraries were not ported) this is why this work started out with Python 3, the more modern version which will be supported for a much longer time frame.

There is however one Python 2 feature which was removed in Python 3 and provides a lot of convenience for PySpark applications: tuple parameter unwrapping.

In Python 2, the following expression is valid:

```
sc.parallelize([(1,2), (2,3), (3,4)])\
    .map(lambda (first, second): first + second)
```

As you can see the lambda has a tuple as an argument, the values passed to it will automatically be unwrapped meaning that the first element of a pair will always be in `first` and the second in `second`. This makes for very readable code even if the elements of a list are be more complicated as parameter unwrapping also works for nested tuples.

The same expression however is not valid Python 3. Instead, one would have to perform the operation like this:

```
sc.parallelize([(1,2), (2,3), (3,4)])\
    .map(lambda pair: pair[0] + pair[1])
```

This can get confusing, especially in more complicated programs where the list might contain tuples of the form `((word, count), somelist)`. Unfortunately there is no real way around this when using Spark's built in functions.

Since lambdas in Python may only consist of one expression it is not possible to do the unwrapping via an initial assignment.

One possible solution is the use of named tuples which allow accessing the members of tuples by name these do however need to be explicitly defined. The other would be to use functions that are defined externally and called from inside the lambda, the tuple can then be unwrapped in the function call.

```
def add_up(a, b):
        return a + b

sc.parallelize([(1,2), (2,3), (3,4)])\
    .map(lambda pair: add_up(*pair))
```

While minor versions of Python do bring some additional features they mostly retain backwards compatibility with code written for prior minor versions. That being said the Python version used for development is 3.5.2.

Summary *With the use of the describe testing and deployment setup we were able to implement the proposed design. A wrapper was implemented around an existing Java part-of-speech tagger to enable efficient tagging in both German and English texts. After discussing the issue and challenges in implementing the proposed design in an efficient manner we can now move on to evaluation of both the performance and the quality of the final results.*

6 Evaluation

This section contains the evaluation of the results generated by the proposed pipelines.
The section starts out with an overview that will be followed by some data exploration. This data exploration will be used as a basis to discuss both quality of results and the performance in Section 6.3 and Section 6.4 respectively.

6.1 Overview

Figure 8 shows what the actual implemented compound pipeline based on the design proposed in Section 4.2 does when executed using Spark. The graphic is taken from Spark's web interface which

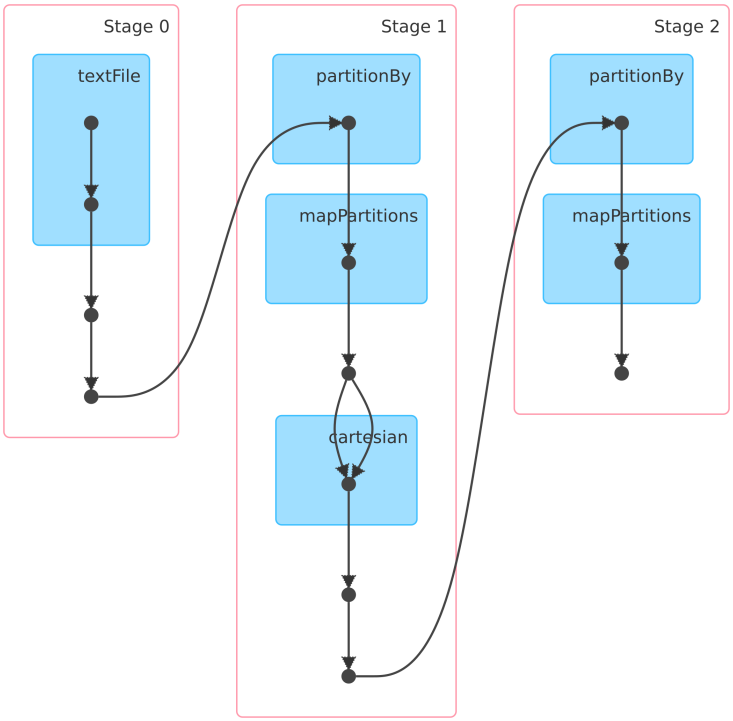


Figure 8: Visualization of the basic compound pipeline as displayed in the Spark web interface

allows visualizing the execution as a DAG.

The graph shows many transformations are which simple map operations on the whole data or a subset of it. When going in detail on the performance analysis, it is important to know that which operations could potentially be the root cause for performance issues. The only really expensive operation in the graph is the Cartesian product. It is of course important, to keep in mind that even any of the map operations might have a large impact on the computation time in praxis.

These considerations will play a part when evaluation the performance of the pipeline.

Unless otherwise stated, the analysis in all of Section 6 is based on 10% of the dataset for the German data or the full data set when the English version is concerned.

6.2 Exploration of Results

6.2.1 New Words Discovered

To assess the performance of the cross-product operation, the section that is likely to be the bottleneck in the compound word pipeline, it is necessary to know how large this operations input is. While the number of articles has some correlation with this operation's expensiveness it is not a good value to look at since the cross product is only done across all *unique* nouns. In other words, the most expensive operations runtime only indirectly depends on the number of articles.

It seems likely that, while nouns are more or less evenly distributed across articles, the number of unique nouns is increasing less and less as more articles are taken into the calculation. As duplicates are however removed early in the computation pipeline (and hence not included in the crossproduct) they should have little impact on the overall runtime and the more relevant metric should be unique nouns. Unique nouns would be assumed to be some asymptotically rising function of the number of articles. The function of unique nouns with the number of articles is expected to be a function of bounded growth.

Figure 9: Number of total words found with number of articles (German).

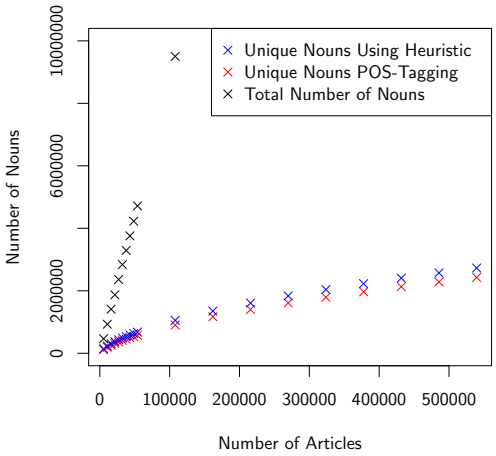
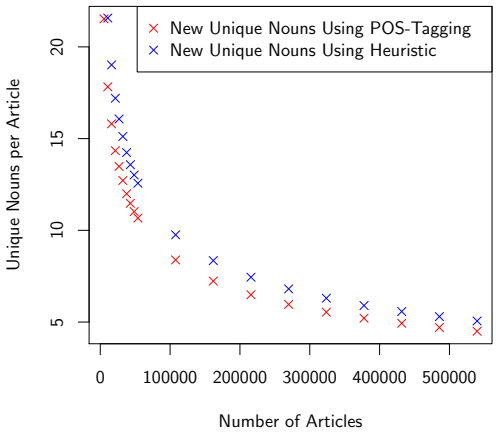


Figure 10: Number of new, unique words discovered with each article (German)



In total, this analysis of the *whole* German Wikipedia yields **2.4 million unique nouns** in around **540.000 articles**. In the dataset of 539,587 articles, a **grand total of 40 million nouns** are found. Only around 2.4 million of those however are unique occurrences of the nouns. There are therefore an average of **4.45 unique nouns per article** when averaging across all articles.

Figure 10 shows that as suspected the more articles are taken into account less unique words are discovered with each article. It can be observed, that the simple heuristic approach works well for finding a large amount of nouns even though it finds, as expected, fewer nouns then the part-of-speech approach.

Figure 11: Number of total words found with number of articles (English)

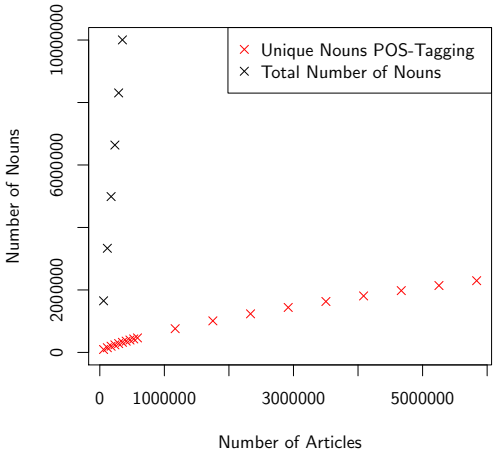


Figure 12: Number of new, unique words discovered with each article (English)

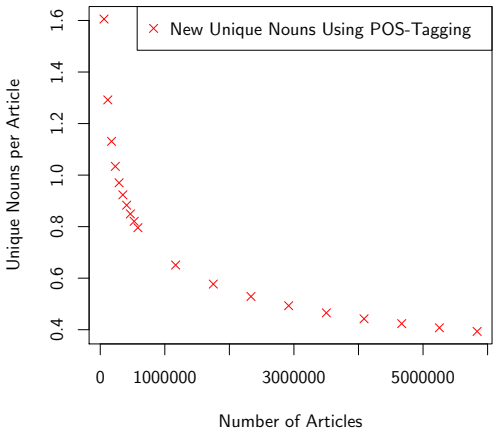


Figure 11 and Figure 12 show the same analysis for the English Wikipedia dataset. For the English dataset analyzing **5.8 million articles** results in finding a total of **2.3 million unique nouns**. This are far fewer unique words per article then the German dataset contains according to the previous analysis.

In general, the same observations done for the German data are also valid, the amount of unique words is just a lot lower.

6.3 Quality of Results

6.3.1 Identifying Nouns

The initial step of identifying nouns in the compound pipeline has a large impact on the quality of the resulting data. Since there is no tagged version of the Wikipedia available it is difficult to evaluate how well this step is working. One way to do it is to look at the resulting data and try to spot either base words or compound words that don't appear to be nouns.

Recall that each resulting tuple of the compound pipeline is of the form:
$((baseword, count), [(compound, count), ...])$

The results ordered by occurrence of the base word can be seen in Figure 13, those when ordering by occurrence of all compound words in Figure 14. All words in these results should be nouns.

Figure 13: First 10 results when sorting by count of base words (German data).

```
('Jahr', 33989), [('Jahrhundert', 13507), ('Jahrhunderts', 9114), ...,('Konvergenzjahr', 1)]
('Jahre', 29849), [('Jahresende', 213), ('Jahreszeit', 189), ...,('Jahrestafel', 1)]
('Jahren', 24804), [('Folgejahren', 324), ('1970er-Jahren', 187), ...,('Normaljahren', 1)]
('Stadt', 24087), [('Stadtteil', 2724), ('Hauptstadt', 2458), ...,('Stadtprojekte', 1)]
('Zeit', 24010), [('Zeitpunkt', 2726), ('Zeitung', 1781), ...,('Diktaturzeit', 1)]
('Ende', 16130), [('Kriegsende', 905), ('Legende', 687),       ('Labor-Day-Wochenende', 1)]
('Teil', 16096), [('Stadtteil', 2724), ('Ortsteil', 2539), ..., ('Seanteil', 1)]
('Mai', 14125), [('Mairie', 67), ('Maigret', 64),   ('Maico-Motorfahrräder', 1)]
('Gemeinde', 13888), [('Gemeindegebiet', 1409), ('Gemeinderat', 1176), ..., ('Gemeindebedienstete', 1)]
('Jahrhundert', 13507), [('Jahrhundertwende', 223), ('Jahrhundertelang', 20), ..., ('Jahrhundertfund', 1)]
```

Figure 14: First 10 results when sorting by accumulated count of compound words (German data).

```
('Ten', 42), [('Staaten', 5204), ('Westen', 3922), ...,('Frauengeschichten', 1)]
('Gen', 122), [('General', 2289), ('Folgen', 1743), ...,('Ortsbeschreibungen', 1)]
('Ver', 3), [('Verein', 3932), ('Verbindung', 3650), ...,('Vergebungsjahr', 1)]
('Sch', 6), [('Schule', 3651), ('Schlacht', 3025), ...,('Schraubenzieher', 1)]
('Ter', 2), [('Kilometer', 4505), ('Tochter', 4336), ...,('Bremsgitter', 1)])
('Ern', 1), [('Einwohnern', 2812), ('Ländern', 2703), ...,('Siedlungskörpern', 1)]
('Ren', 13), [('Jahren', 24804), ('Verfahren', 2724), ...,('Elevatoren', 1)])
('Tung', 11), [('Bedeutung', 6857), ('Richtung', 4692), ...,('Organdurchblutung', 1)]
('Men', 21), [('Menschen', 9788), ('Unternehmen', 6114), ...,('Pflanzenvorkommen', 1)]
('Hen', 1), [('Menschen', 9788), ('Sprachen', 2327), ...,('Mini-Sternchen', 1)]
```

For Figure 13 it is obvious that a lot of words are correctly identified as nouns, all base words appear to be nouns.

For the second sample of results this seems far less clear cut. In fact, a none of the base words in this sample appear to be common nouns. The compound words appear to all be nouns compounds however all appear to be nouns.

There are some other issues with the words but these are not related to noun identification and will be addressed later on.

A lot more false positive nouns appear to show up in the bottom sample. In fact, on first sight 9 out of 10 (everything but "Tung" which is at least a proper noun) don't appear to be German nouns.

What might the reasons for this be? The bottom sample prefers words that appear in more compounds, and the words that appear in the most compounds will always be false positives like "Ern" which matches a lot of plural nouns. This does not answer the question of why these short words were recognized as nouns.

When investigating this a few of the issues can be found, in fact many of the words have single digit occurrences across this (10% of the German Wikipedia) sample. This makes investigating by searching for the words easy to do.

One of the issues that could be identified are names which can not reliably be recognized as such. "Tung" for example is a unit of measurement used in Sumatra, discerning proper nouns from common nouns is clearly hard for the part-of-speech tagger to get right all cases. After all they can not be recognized by syntactic features of the sentence as they can be used interchangeably with common nouns. In English texts, there is at least the distinction of proper nouns being capitalized but depending on the context this might also not be an indication. An approach with more information would probably improve these results, using existing datasets like a dictionary could identify almost all common nouns. The goal of this work however is to work without such prearranged datasets.

In Section 4, the choice was made to not include proper nouns in the search for compounds. Words like 'Tung' violate this goal, it is however exceptionally difficult for the part-of-speech tagger to distinguish between proper and common nouns for words that are not known.

Figure 15: First 10 results when sorting by count of base words (English data).

```
(('time', 2110089), [('lifetime', 72526), ('maritime', 32650), ('overtime', 32442), ('meantime', 25712), ('wartime',
    23418), ('timeline', 21666), ('halftime', 17336), ('timetable', 12044), ('half-time', 10180), ('peacetime', 8408)
    , ('timeslot', 8072), ('primetime', 7922),
(('year', 1504785), [('yearbook', 8246), ('yearning', 3314), ('year-end', 2524), ('half-year', 1412), ('yesteryear',
    556), ('midyear', 296), ('yearling', 212), ('year-and-a-half', 202), ('light-year', 154), ('mid-year', 106), ('
    schoolyear', 102), ('school-year', 52), ('bir
(('part', 1448314), [('partner', 205650), ('partnership', 146584), ('participation', 116138), ('particle', 45766), ('
    counterpart', 36214), ('participant', 34432), ('partition', 31146), ('parting', 6766), ('participle', 4010), ('
    rampart', 3472), ('partitioning', 2854), ('pa
(('season', 1345619), [('postseason', 21214), ('preseason', 16666), ('off-season', 12332), ('offseason', 9158), ('mid-
    season', 6996), ('pre-season', 4928), ('midseason', 2626), ('seasoning', 2070), ('post-season', 1422), ('
    seasonality', 840), ('half-season', 784), ('season
(('team', 1263933), [('teammate', 21183), ('team-mate', 4896), ('first-team', 2483), ('teamwork', 2411), ('team-up',
    291), ('teamster', 210), ('fireteam', 89), ('second-team', 83), ('tag-team', 80), ('loveteam', 69), ('superteam',
     63), ('team-work', 38), ('teamed-up', 38),
(('film', 1231330), [('filmmaker', 18364), ('filmmaking', 5903), ('filmography', 1406), ('film-maker', 1346), ('
    microfilm', 1311), ('filming', 939), ('film-making', 855), ('filmation', 675), ('biofilm', 646), ('telefilm',
    598), ('sound-on-film', 274), ('filmfare', 227), ('
(('album', 1169924), [('mini-album', 3890), ('double-album', 508), ('solo-album', 164), ('debut-album', 80), ('
    albuminuria', 78), ('studio-album', 74), ('live-album', 66), ('full-album', 48), ('concept-album', 40), ('
    minialbum', 32), ('studioalbum', 28), ('albumcover', 26)
(('school', 1137936), [('schooling', 32680), ('schoolhouse', 9056), ('schoolteacher', 8152), ('high-school', 6480), ('
    schoolboy', 6082), ('schoolmaster', 5514), ('schoolgirl', 2404), ('schoolmate', 1634), ('schoolroom', 1306), ('
    pre-school', 1250), ('schoolwork', 870), ..., ]
(('name', 1131865), [('surname', 165068), ('nickname', 78762), ('namesake', 22382), ('codename', 5522), ('nameplate',
    3798), ('namespace', 2270), ('filename', 1816), ('placename', 1696), ('pen-name', 1688), ('username', 1010), ('
    forename', 972), ('place-name', 828), ...]
(('family', 1128298), [('subfamily', 43184), ('superfamily', 11204), ('single-family', 1112), ('sub-family', 454), ('
    font-family', 226), ('stepfamily', 178), ('pro-family', 158), ('work-family', 104), ('family-drama', 82), ('
    jupiter-family', 66), ('step-family', 54), ...]
```

Figure 16: First 10 results when sorting by accumulated count of compound words (English data)

```
(('ion', 16390), [('population', 1757526), ('station', 1354476), ('production', 912398), ('version', 897898), ('region
    ', 888178), ('television', 878662), ('position', 825090), ('addition', 770252), ('election', 769986), ('education
    ', 760800), ('construction', 678060), ('in
(('tion', 48), [('population', 1757526), ('station', 1354476), ('production', 912398), ('position', 825090), ('addition
    ', 770252), ('election', 769986), ('education', 760800), ('construction', 678060), ('information', 648864), ('
    competition', 582986), ('section', 552390),
(('ent', 218), [('government', 1735774), ('development', 941262), ('president', 576282), ('movement', 466704), ('
    student', 465026), ('tournament', 451704), ('management', 403706), ('department', 326440), ('percent', 300292),
    ('treatment', 286226), ('settlement', 275656), (
(('nce', 43), [('performance', 576750), ('science', 441148), ('appearance', 418120), ('evidence', 385242), ('conference
    ', 368430), ('experience', 326516), ('province', 318688), ('influence', 299918), ('presence', 255232), ('distance
    ', 254702), ('audience', 222360), ('refer
(('ity', 48), [('community', 990100), ('municipality', 464418), ('majority', 401180), ('university', 369358), ('
    security', 294468), ('ability', 292802), ('quality', 291522), ('density', 289142), ('capacity', 283666), ('
    activity', 275930), ('facility', 233476), ('authority'
(('ation', 50), [('population', 878763), ('education', 380400), ('information', 324432), ('organization', 245538), ('
    location', 209898), ('operation', 167433), ('administration', 136563), ('formation', 135404), ('association',
    132173), ('creation', 121605), ('situation', 1
(('ing', 5119), [('building', 576487), ('training', 302907), ('living', 182866), ('beginning', 155679), ('opening',
    154109), ('meeting', 153699), ('housing', 149988), ('morning', 137077), ('spring', 134444), ('something', 128063)
    , ('recording', 125371), ('painting', 109829
(('con', 12366), [('construction', 678060), ('control', 661398), ('contract', 451788), ('conference', 368430), ('
    concept', 273022), ('content', 233368), ('concert', 225340), ('conflict', 200862), ('condition', 194262), ('
    connection', 192120), ('contact', 178306), ('constit
(('ter', 1453), [('character', 681770), ('daughter', 636166), ('center', 541702), ('writer', 397636), ('sister',
    336142), ('master', 323110), ('computer', 294898), ('territory', 275868), ('letter', 268132), ('minister',
    211898), ('winter', 201150), ('matter', 198854), ('qu
(('sion', 83), [('version', 897898), ('television', 878662), ('division', 625680), ('decision', 369964), ('mission',
    309368), ('expansion', 204400), ('session', 162344), ('invasion', 159556), ('extension', 150200), ('commission',
    139178), ('expression', 131514),
```

Both Figure 16 and Figure 15 show that the same problems occur with the German dataset.

There are no issues with the base-words in the first dataset, the second shows the same issue of having a large number of words that are not actually nouns. The nouns in the second English sample occur a lot more often than those in the German sample, this is because only a tenth of the German dataset is analyzed while the English dataset is larger to begin with.

These results paint a very poor picture of the algorithm. Many of these false results would probably not even have a large impact on the following results. Yet it is still a good idea to get rid of as many of the compounds that aren't real compounds as possible, both to improve the quality of results and even to help reduce the file's size for easier handling.

The full results file for the English Wikipedia totals only 121 MByte making the post processing to improve the results very convenient as opposed to recomputing the whole result which would take a long time (see Section 6.4).

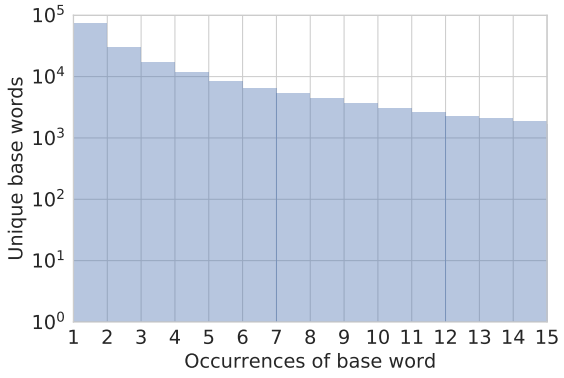
Filtering To improve specifically the results from Figure 16 and Figure 14, it is a good idea to consider that most of these nouns only occur very rarely in the dataset. This is the case because they are either misclassified by the part-of-speech tagger in very few cases (and are not actually nouns) or because they are words that occur very rarely. The letter might even be the case due to a misspelling of a known word.

It seems like a good idea to remove some portion of rare words. There is a choice to be made about large of a section of words should be removed.

The simplest approach would be to remove any base-word that has less than a specific constant threshold of occurrences. An approach of constant numbers might work but would scale poorly to larger datasets. It appears best to find a cutoff point in terms of number of times a word occurs, removing a set percentage of words that occur very rarely.

When looking at the distribution of the occurrences of base words, it would be expected that there are some base words that occur often while the majority of words occurs relatively rarely. This is expected as there are a lot of words that are only relevant in certain contexts and therefore get used a lot less. The two extremes would be words that only show up once, like some of the false positives or words that show up in almost every article. At some point a word is infrequent enough to not be relevant for further analysis, a word that only occurs say ten times in the whole Wikipedia is very unlikely to be known by someone taking the task. The question is where exactly to set the cutoff point.

(a) English Wikipedia dataset



(b) German Wikipedia dataset

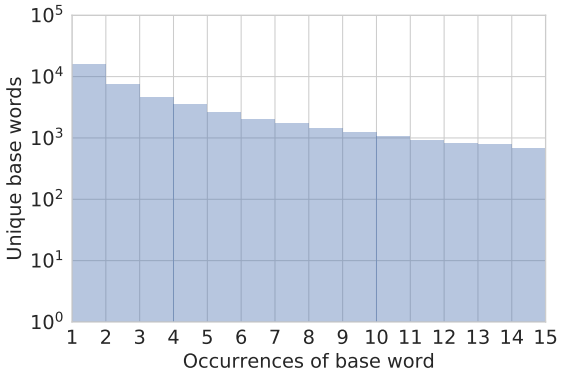


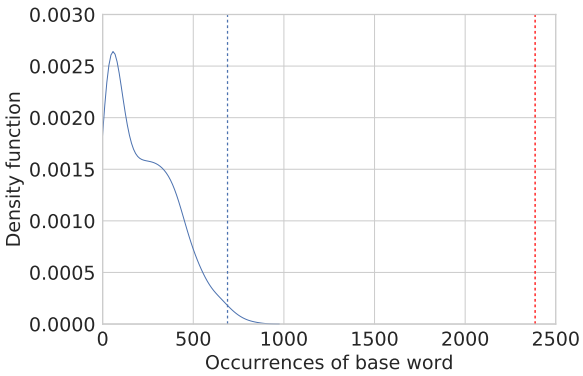
Figure 17: Histogram of number of base word occurrences.

The plot Section 6.3.1 shows a histogram of the base word counts as they occur in the dataset. In the English dataset there are almost 10^5 base words which only occur once in the entire dataset.

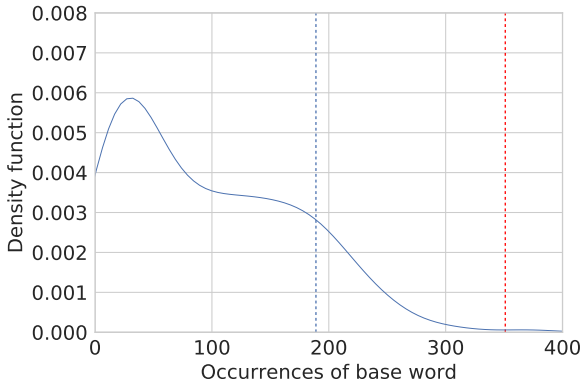
The plot has a logarithmic scale. The number of unique base words per occurrence count is therefore decreasing exponentially. It appears logical that any word with only 1 occurrence should be removed as it might very well just present an error in the data. The plots cuts of after 15 base word occurrences, the entirety of the distribution is visualized in Section 6.3.1.

Figure 18: Density plot of number of base word occurrences

(a) In the complete English Wikipedia dataset



(b) In the complete German Wikipedia dataset



Section 6.3.1 shows how the occurrences of base words are distributed. This plot is cut of at 25000 visually there is no more information to be obtained.

The two lines indicate the 95th and 97th and percentile of the data, it appears reasonable to cut off anywhere around the huge drop in the graph to still retain most relevant words while sorting out a lot of very uncommonly used words. Cutting of before the drop means favoring the retaining rare words over potentially dropping them but cleaning up more wrong results.

Table 3: Percentiles and the number of occurrences needed for a word not to be discarded. German data on the left, English on the right.

Percentile	Minimum Number of occurrences
80th	25
95th	189
96th	249
97th	351
97.5th	429
Percentile	Minimum Number of occurrences
80th	22
95th	689
96th	1222
97th	2386
97.5th	3518

For the English dataset 95% of samples have occurrences of less then 689. When looking at the data at this means that words such as:

- cutout
- cognate

remain in the dataset. Words these however are just about discarded:

- awardee
- tallow
- griffin

Setting the cutoff point at the 97th percentile would cut off such words as: "showroom", "skit" and "carpentry", the 95th percentile seems like the better choice. The 95th percentile appears to be a reasonable compromise as some of the words would be considered rare but not exceedingly so and might come up in CRAs.

Since many of those words will not be known to anyone it seems like a reasonable choice at least for the English dataset to just use the top 5% of most occurring words and thereby clean the results up a lot. In the full English dataset, the output consists of **226972** base words and their respective compounds. After removing 95% of the least occurring base words the resulting dataset accordingly only contains **11349** base-words.

For the German dataset this percentile puts the cutoff point at 189 discarding such words as: "Konven", "Bauch" and "Hängen". These seem like reasonable words to discard so we use the same cutoff point.

Only the second list of words from above for each language changes with this cutoff as the most common compounds are not affected. The results now look as follows:

Figure 19: First 10 results when sorting by accumulated count of compound words (English data) after filtering.

```
(('ion', 16390), [('station', 1354476), ('production', 912398), ('region', 888178), ('position', 825090), ('election',
     769986), ('construction', 678060), ('section', 552390), ('action', 466868), ('collection', 441984), ('edition',
     381718), ('mission', 309368), ...]
(('ing', 5119), [('building', 576487), ('training', 302907), ('opening', 154109), ('meeting', 153699), ('morning',
     137077), ('recording', 125371), ('painting', 109829), ('engineering', 99216), ('teaching', 94027), ('meaning',
     92049), ('funding', 89760), ('singing', 77478), ...])
(('sea', 214844), [('season', 2691238), ('seating', 23286), ('seafood', 10656), ('seaplane', 8060), ('seawater', 7604),
      ('seaman', 6024), ('seaport', 5968), ('seaboard', 4492), ('seaweed', 4482), ('undersea', 3966), ('seashore',
     2678), ('seafloor', 2604), ('seafront', 2446), ...])
(('ship', 250091), [('relationship', 382488), ('championship', 346106), ('leadership', 262752), ('township', 164372),
     ('membership', 157176), ('partnership', 146584), ('ownership', 132710), ('scholarship', 101788), ('friendship',
     75766), ('shipping', 60496), ...])
(('ter', 1453), [('sister', 336142), ('master', 323110), ('letter', 268132), ('minister', 211898), ('winter', 201150),
     ('matter', 198854), ('chapter', 141238), ('painter', 135408), ('terminus', 96022), ('charter', 91370), ('latter',
      59516), ('terrain', 55798), ...])
(('hip', 45221), [('relationship', 382488), ('championship', 346106), ('leadership', 262752), ('township', 164372), ('
     membership', 157176), ('partnership', 146584), ('ownership', 132710), ('scholarship', 101788), ('friendship',
     75766), ('citizenship', 60422), ...])
(('con', 12366), [('contract', 451788), ('content', 233368), ('concert', 225340), ('contact', 178306), ('contest',
     137540), ('context', 130562), ('conversion', 90846), ('conjunction', 65678), ('consequence', 61058), ('
     configuration', 60520), ('confusion', 60002), ...])
(('comp', 866), [('company', 1701922), ('companion', 57296), ('compass', 12644), ('compact', 2168), ('compaction',
     1652), ('complicit', 1388), ('comparability', 826), ('compere', 480), ('compactor', 268), ('compunction', 258),
     ('compacting', 170), ('compline', 104), ...])
(('age', 823836), [('damage', 236004), ('average', 221524), ('message', 139198), ('coverage', 126794), ('percentage',
     109992), ('passage', 109198), ('footage', 73154), ('package', 70836), ('voltage', 47902), ('garage', 44892), ('
     drainage', 42432), ('lineage', 33596), ...]
(('son', 558947), [('season', 1345619), ('person', 234070), ('grandson', 37021), ('poison', 15548), ('sonnet', 3627),
     ('stepson', 2411), ('unison', 2232), ('parson', 743), ('godson', 647), ('telson', 238), ('stetson', 117), ('
     gleeson', 116), ('sonship', 100), ('damson', 87), ...]
```

Summary *The German list shows the issue of plural nouns and different forms of the same noun being included. A potential way to solve this via stemming will be discussed in the next section.*

Overall most of the nouns appear to be correctly identified such. The problems lie with some occasional false-positives. Filtering was able to discard a lot of those but will always also affect some legitimate results.

6.3.2 Compound Pipeline

While the previous section merely focused on the detection of nouns this section will go over the entire output. For evaluating the quality of results it is useful to look at the words that are most likely to not be good results. To do this you can look at the base words that themselves don't necessarily occur often but occur in many compounds words. This favors nouns that happen to also be endings of words and similar mistakes.

When considering the compound words in Figure 19 and Figure 20, another issue becomes apparent that was glossed over in Section 6.3.1. With many base words there are a number of compound words which while starting or ending with the base word don't actually form a valid compounds with it.

Figure 20: First 10 results when sorting by accumulated count of compound words (German data) after filtering.

```
(('Jahr', 33773), [('Jahrhundert', 13470), ('Jahrhunderts', 9075), ('Frühjahr', 1715), ('Jahrzehnten', 969), ('
    Jahrzehnte', 739), ('Jahrhunderte', 725), ('Lebensjahr', 489), ('Jahrhunderten', 475), ('Vorjahr', 285), ('
    Jahrhundertwende', 222), ('Schuljahr', 212), ...]
(('Jahre', 29624), [('Jahresende', 211), ('Jahreszeit', 181), ('Jahrestag', 170), ('Jahreszeiten', 128), ('Lichtjahre',
     126), ('Jahreszahl', 114), ('Lebensjahre', 94), ('Jahreswechsel', 86), ('Jahresniederschlag', 69), ('
    Jahresbeginn', 60), ('Jahreswende', 51), ...]
(('Jahren', 24521), [('Folgejahren', 323), ('Lebensjahren', 105), ('Anfangsjahren', 98), ('Nachkriegsjahren', 73), ('
    Lichtjahren', 65), ('Kriegsjahren', 51), ('Schaltjahren', 39), ('Vorjahren', 29), ('Regierungsjahren', 25), ('
    Dienstjahren', 22), ('Jugendjahren', 13),
(('Stadt', 23901), [('Stadtteil', 2703), ('Hauptstadt', 2450), ('Stadtgebiet', 949), ('Altstadt', 896), ('Innenstadt',
    877), ('Stadtteile', 573), ('Stadtrat', 555), ('Kreisstadt', 538), ('Heimatstadt', 512), ('Kleinstadt', 391), ('
    Stadtbezirk', 388), ('Stadtzentrum', 3
(('Zeit', 23335), [('Zeitpunkt', 2680), ('Zeitung', 1726), ('Zeitschrift', 1586), ('Zeitraum', 1194), ('Spielzeit',
    961), ('Amtszeit', 941), ('Folgezeit', 691), ('Zeitungen', 653), ('Neuzeit', 552), ('Hochzeit', 542), ('
    Zeitschriften', 483), ('Blütezeit', 467), ('Regie
LINE (('Ende', 15920), [('Kriegsende', 905), ('Legende', 677), ('Vorsitzende', 557), ('Saisonende', 403), ('Wochenende
    ', 251), ('Jahrhundertwende', 222), ('Jahresende', 211), ('Tausende', 180), ('Studierende', 171), ('
    Jahrtausendwende', 152), ('Reisende', 148), ('Lebensende
(('Teil', 15742), [('Stadtteil', 2703), ('Ortsteil', 2508), ('Großteil', 1350), ('Bestandteil', 1302), ('Teilnehmer',
    1033), ('Vorteil', 985), ('Teilnahme', 904), ('Teilung', 494), ('Teilchen', 450), ('Nachteil', 388), ('
    Teilnehmern', 319), ('Gegenteil', 307), ('Teilst
(('Gemeinde', 13830), [('Gemeindegebiet', 1409), ('Gemeinderat', 1173), ('Kirchengemeinde', 571), ('Verbandsgemeinde',
    441), ('Ortsgemeinde', 329), ('Landgemeinde', 327), ('Marktgemeinde', 299), ('Samtgemeinde', 274), ('
    Gemeindevertretung', 253), ('Gemeindegebiets', 23
(('Jahrhundert', 13470), [('Jahrhundertwende', 222), ('Jahrhundertelang', 20), ('Vierteljahrhundert', 19), ('
    Jahrhunderthälfte', 16), ('Jahrhunderthochwasser', 13), ('Jahrhundertmitte', 10), ('Jahrhunderthalle', 7), ('
    Jahrhundertwert', 5), ...])
(('Namen', 10992), [('Ortsnamen', 331), ('Beinamen', 313), ('Spitznamen', 298), ('Vornamen', 290), ('Namensgeber', 255)
    , ('Familiennamen', 214), ('Markennamen', 169), ('Namensgebung', 160), ('Straßennamen', 144), ('Personennamen',
    142), ('Künstlernamen', 138), ...])
```

This problem is especially apparent in the word 'ion'. 'Population' for instance is not a compound of 'populat' and 'ion' but instead just a single word 'population'.

There is one potential fix to this issue. A lot of these could be filtered out in our computation by ensuring that each compound word actually consist of two known words. This can be done in a post processing step checking for each word if it is present in a dictionary.

The technique above of filtering by words that don't consist of two valid words could be performed using a list of words from the Wikipedia. For simplicity this work is going to rely on a dictionary for now. It would however relatively simple to generate a word list based on the Wikipedia dataset.

For the word 'ion' for ensuring that each of the other words is also a valid word using a dictionary returns this data:

```
(('ion', 16390), [('production', 912398), ('position', 825090), ('election', 769986), ('construction', 678060), ('
    division', 625680), ('section', 552390), ('action', 466868), ('collection', 441984), ('edition', 381718), ('
    mission', 309368), ...]
```

There still is the issue of some words seemingly forming a compound word, while actually having a different origin. One such example would be the word 'section', while both 'sect' and 'ion' are valid words 'section' is not in most contexts actually a compound word of the two words. This ambiguity seems impossible to resolve without a deeper understanding of the semantics of the words. It does however seem reasonable to just assume any word that shows up a lot more then it's base word to be a false positive. In theory, it is possible for the compound to be more common then the individual word, it does however seem unlikely. This might however have little to no influence on the overall results when generating an solving CRAs.

```
(('opposition', 112929), [('oppositionism', 6), ('oppositionafter', 2), ('oppositionfrom', 2), ('oppositionand', 2)])
```

The above word shows three compound with stop words, it is unclear how exactly these results where produced, possibly there was some issue with handling of the wiki-markup language leading to missing white spaces. It might however be as banal as being a spelling mistake inside the Wikipedia, this can never be completely avoided with such a large dataset. Filtering to remove compounds with stop words is performed.

This result would probably not impact the solving of CRA tasks, as the words in this example are rare and would therefore get a very low score. The only motivation to remove them would be to not create a task with "opposition".

Unfortunately improving these specific results further seems infeasible. Each of the words could conceivably be an actual compound used in natural language. Take the 'mission' for example an 'miss-

ion' might be an ion that missed some target. Without gaining significant semantic understanding of the sentences a word appears in it appears infeasible to understand which of these words are no actually a combination of the two words but instead form an independent word.

For an additional issue we again consider the original German dataset from Figure 20. Even after performing the cut off of very rare words there is still one other issue that wasn't apparent in the English dataset. The word "Jahre" appears both as "Jahre" and as "Jahren". This is undesirable as only "Jahre" should actually show up in a CRA. This problem can be solved via stemming, a technique that is used to reduce a word to it's stem.

Stemming is among the functionality that is offered by the `nltk` package. Using this on base words and merging words that are equal after the operation results in this updated list for the word "Jahr":

```
(('jahr', 91611), [('Jahrhundert', 13470), ('Jahrzehnten', 969), ('Jahrzehnte', 739), ('Jahrhunderte', 725), ('
    Lebensjahr', 489), ('Jahresende', 422), ('Jahrzehnt', 191), ('Jahreszeit', 181), ('Jahrtausend', 177), ('
    Lichtjahre', 126), ('Geschäftsjahr', 119), ...])
```

The entry was created by joining those for "Jahren", "Jahre" and "Jahres".

One possible additional measure could be removing any base word that has very few compound word occurrences.

Ambiguous compound words combined with the issues in identifying nouns mean that the results will never be perfect, but overall some post processing after the compound pipeline was able to improve the quality of the results.

All of this proposed additional filtering techniques will be used for compound generation and solving.

6.3.3 Generating Compounds

There were two metrics proposed which are combined to assess the relevance of a word in compounds with a given base word. These two criteria aim to balance a word's frequency in context with the base word with the uniqueness of a word (since a unique word is a better choice for identifying the base word). In the simple implementation for each base word a CRA is generated by taking the three highest scoring compound words for that base word.

One measure for assessing the quality of results would be to compare them to the existing CRA tasks. This means checking whether or not the proposed methods come up with the same compound words for a given base word as the published lists show. Since the criteria for creating a CRA are not objective one should not expect the results to line up perfectly. Instead, it should be perfectly acceptable to, for each task, have the words in the published list show up in the first 10 results of the generated list.

As all solutions that are not actually nouns are not available in the set of results (as base words) there is an initial failure rate. In fact, 16 of 144 words don't show up as base words, they are mostly adjectives:

- sore
- common
- blind
- fast
- sick
- short
- grand
- under
- full
- super
- sweet
- false
- after
- brown
- straight
- broad

Table 4 shows how many of the three challenge words can be found in the first n words of the list for a given base word (with the base word being the solution for the three challenge words) in the result set of the compound pipeline. When disregarding those base words that don't show up

at all in the result, the average number of words that are the same as in the original task is higher. The words that don't show up all violate the assumptions made in the design.

Table 4: Quality of generated English CRAs

Initial n results considered	Average number of words found	
	With unavailable base words	Without unavailable base words
all	1.93	2.17
20	1.27	1.43
10	1.04	1.17

When disregarding the base words that could not be found the values are very high. There seems to still be some room to improve here!

Ultimately, the fact that very few words are found in the few highest scoring words only proves that either the proposed algorithm doesn't line up with the one used in Bowden & Jung-Beeman's paper or that there is no deterministic algorithm used in the paper.

The fact however that only about two thirds of all words show up anywhere in the complete lists is concerning. This does indicate that our sample of natural language is imperfect, perhaps compound words separated by spaces are not included in this dataset.

Manually reviewing a section of the results also shows that they are at least to some degree reasonable. As this is the initial approach discussed in Section 4 it is not guaranteed that the best fitting solution is in fact the one listed.

Table 5: Generated English CRA (with initial technique)

Challenge Words			Solution
womanhood	baseball	cation	career
food	men	man	service
case	biz	runner	show
shed	fresh	waste	water
sub	include	micro	district
country	hill	kick	side
grand	step	god	father
rail	high	run	way
jury	pent	bad	award
ontology	inane	photo	period

There are still many problems in this data, for example the `grand/step/god` task is very ambiguous, the answer could just as well be daughter or mother. While the pruning routine makes sure no two different words appear as answers for the same three challenge words, it doesn't make sure that they are actually uniquely identifiable. While these results might not all be perfect it seems that at least using some manual review they could quickly be turned into a high quality list. This is a random sample from a 5335 line file so there is a large pool of potential tasks to draw from when using manual review.

The fact that the results are not better could also be attributed to the fact that the pipeline does not yet properly handle English compound words that are separated by a space. If this is the only issue results with a German dataset might be much better.

For this dataset 41 out of 130 base words are not found at all. This is not unexpected as the German dataset is much smaller. What is surprising however is that there is a lot less words generated that coincide with the existing dataset. The most probable reason for this is that the German dataset is just not large enough.

We can take a look at this example of generated words and the words in the existing dataset:

Table 6: Quality of generated German CRAs

Initial n results considered	Average number of words found	
	With unavailable base words	Without unavailable base words
all	0.7	1.09
20	0.37	0.55
10	0.31	0.21

```
Existing Task: oase/mann/lohn steuer
Generated list: ['einkommen', 'pflichtigen', 'einnahmen', 'pflichtige', 'umsatz',
            'wagen', 'mehrwert', 'erbschaft', 'kraftfahrzeug', 'schenkung']
```

It is obvious that the generate list shows reasonable words, they just happen to not match the ones that are in the original task. It is quite possible that this might improve with a larger dataset.

Here is a sample of some German CRAs that were generated:

Table 7: Generated German CRA (with initial technique)

Challenge Words			Solution
stern	sinn	schirme	bild
statt	zeuge	netz	werk
freundes	wahl	landtagswahl	kreis
zeugen	blätter	lotsen	flug
gemeinde	einzugs	naturschutz	gebiet
bundes	vasallen	mitglied	staat
stifts	pauls	kloster	kirche
urheber	ehrenbürger	pfand	recht
welt	vize	rekord	meister
alter	speerwurf	fechter	nation
voll	langer	schwingen	hand

Summary Overall the generate CRA tasks while not lining up with existing, published tasks appear to be somewhat reasonable. The fact that some compound words don't show up in the lists at all indicates that the dataset might still be too small, even the full English one.

6.3.4 Generating CRAs - Alternative Technique

While the implementation a good approach in generating CRA it does not guarantee anything regarding the correctness of the presented solution. To truly guarantee this the results have to essentially be cross checked for every base word. This is very slow and the main disadvantage of this second technique. In fact, for the English data this did crash due to some memory limitation after about 10 hours on the cluster. For the smaller German dataset it did however work and produced these results:

It was a subset of just a thousand German base words. When looking in detail, these CRA tasks appear less ambiguous than those using the simple method. This is however hard to evaluate properly.

6.3.5 Solving Compounds

The second proposed method for generating compounds, barring any implementation issues, always produces tasks that can automatically be solved correctly in 100% of cases.

Focusing on the initial, simpler method we get the following results:

Table 8: Generated German CRA (with advanced technique)

Challenge Words			Solution
rändern	viertel	schur	wald
rechtsform	kosten	futtermittel	zusatz
wort	satz	buchstaben	folge
baren	weiten	welt	sicht
grüne	ausbringen	bechern	silber
leid	betreff	mitarbeit	enden
container	seiten	handels	schiffe
panzer	volks	rebellen	führer
flinten	wunden	armbrust	schuss

Real World Tasks - English Of the 144 English compounds we discard those were the solution is not in our dataset, as they all violate our assumptions (the solution is not a noun) which leaves us with 129 CRAs.

In 97 cases, the word is in the 15 best scoring solutions, in 28 of those it is the first result. The task is correctly solved if the first result is the correct one.

We already have a 21.7% chance of solving a given compound just by using the total score of the found words. Solving one in five tasks is not that bad. As 97 of 129 tasks where correctly solved in the first 15 solutions it seems reasonable that at least some of those could be solved chaining the scoring if not all words occur in the compound words.

This can be improved drastically by prioritizing any solution where all three words match. This way 44 of the words 129 can be found meaning a success rate of about 34%.

Real World Tasks - German For solving German words we check against a dataset of 130 words. Unfortunately, the correct word is found with any score in just 10 of 130 words. It is the first option in just a single case!

This is obviously not a desired result. There are two conceivable reasons for this.

1. The German Tasks are harder, they require more language to be known
2. It is merely the fact that the German dataset is this small

After the filtering the German dataset consist of merely 3,000 lines while the English dataset still has over 11,000 base word entries. The published German tasks might also be more difficult, in the sense that the involved words occur more rarely. When considering the time taken by humans in the studies however it the difficulty seems comparable [2] [8].

6.3.6 Semantic Grouping

To asses the quality of the semantic grouping we will look at the compounds with the word game for now. In Figure 5, some possible semantic relations were laid out by hand. The analysis will now be split into two parts, first assessing the quality manually by checking if the groups make any sense. After that the groups from Figure 5 are used to check if there is an overlap with them.

First we consider a very simple example of German words. Using the idf approach as discussed in Section 4 we get the distance vector in Table 9 for the word 'Haft' (imprisonment). Where the distances indicates how far on average the word is removed from the next occurrence of the word 'Haft'. It shows a lot of adjectives that certainly are related to the word. Among them "bestrafte" (*punished*) and "mehrmonatig" (*for several months*), all words appear to really be related to the word. This indicates that the vector does in fact somehow characterize the word.

Table 9: A wrapped table going nicely inside the text.

Word	Distance
bestrafte	0.26
mehrmonatiger	0.29
jahrelanger	0.30
lebenslanger	0.33
herausstellte	0.35
anzutreten	0.36
lebenslange	0.38
beraten	0.39
verstorben	0.39
umgewandelt	0.52
entlassen	0.53
verurteilt	0.54
bestätigt	0.59
gemeinschaftlichen	0.64
frühestens	0.72
dreieinhalb	0.73
heimtückischen	0.80
ersetzte	0.96
vorübergehend	1.00
kurzzeitig	1.02

A list of words like this can be built for all compound words containing 'Haft'. For each such compound word this generates a vector of distances which words have. It can be assumed that this vector does in some way represent semantics of the compound word. With this assumption one can cluster the compound words using the distance vectors as feature vectors.

The list below was generated for different compound words with 'Haft' using k-means with $k = 2$

```
Schutzhaft: 1
Haftentlassung: 1
Einzelhaft: 1
Untersuchungshaft: 0
Mannschaft: 0
Ortschaft-7: 0
Schwangerschaft: 0
Gesellschaft: 0
```

K-means with $k = 3$ results in the following list.

```
Schutzhaft: 0
Einzelhaft: 0
Untersuchungshaft: 0
Haftentlassung: 0
Mannschaft: 1
Ortschaft: 1
Schwangerschaft: 1
Gesellschaft: 2
```

As we can see those words actually related to imprisonment (Schutzhaft, Haftentlassung and Untersuchungshaft) tend to end up in one cluster. These results indicate that there is at least some merit to the clustering approach and to using the distance of surrounding words as an indication of the words semantics.

The problem with using K-means for this is that no domain knowledge is present to find out which value for k is the most reasonable. An approach to solving this is to plot the development of

the mean square error as more clusters are added. If from a certain point on, the error is decreasing a lot less this value for k can be assumed to be a good representation of the number of clusters the data naturally fits into. This is called the elbow method[6].

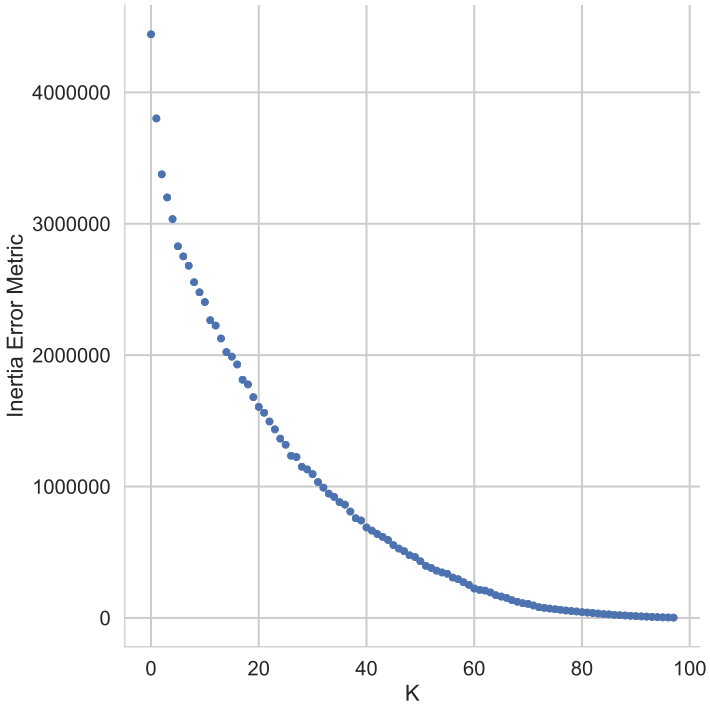
Let's apply this technique to the word game for which there was theoretical example of in Section 4. On a subset of words for $k = 3$ these clusters are generated:

```
Cluster 0:          Cluster 1:          gametime
gamezone            gameleste           gameness
pregame             ballgame            minigame
endgame             wargame             gameboard
gameshark           gameday             gamelyn
gamepad             gameworld           gameover
gamelan             gamekeeper          gamebird
videogame           gameforge           gamesphere
gamespy                                 gamecock
gameplay            Cluster 2:          gamefishing
gameshow            gamemasters         gameplaying
gamecube            gameboy             gamebook
gamepro             amalgame            metagame
postgame            subgame             gamecity
                    gamemaster
```

As you can see there is some relation to the theoretical clusters as laid out in Figure 5.

"Pregame" and "endgame" already end up in the same cluster, the same goes for "gamepad", "gamelan" and "videogame", it seems however that a greater number of clusters would be much more suitable for this task. To find out exactly which number is suitable we plot the mean squared error and look for an elbow.

Figure 21: Error Metric for different K values.



The error plot shows no clear elbow indicating that the choice of clusters is ambiguous which makes sense given that the whole data could be sorted into different clusters that all make sense.

Let us instead consider a subset of only the data presented in the example in Figure 5.

6.4 Performance Compound Pipeline

When implementing the proposed design for this specific dataset the only real runtime requirement is to stay within a reasonable time frame for all computations. It is, however, still desirable to achieve better performance for several reasons:

1. quickly iterating on parameters or implementation details
2. applying the same process to larger datasets
3. academic interest

Spark's aim is to achieve good performance and yet involve little complexity for the programmer. This section aims to evaluate how well this approach works for the task at hand.

We will focus on the compound pipeline which creates the list of base words with their compounds from the raw Wikipedia data. The total runtime of the compound pipeline for the English dataset was: 208 hours and 39 minutes. It is clear that while this might be acceptable for a one time execution it would be preferable especially when iterating on the design to improve this runtime. With the current setup it is not feasible to rerun the whole pipeline after, for example, finding a minor bug.

Any of the post processing that is performed the results of this pipeline does not require performance analysis. As it finishes, even on a single machine in a few minutes at a most. The only exception being the advanced compound generation. The resulting compound file for the English dataset for example is only 121 MByte, evaluating the score of each base word and similar tasks can be done very quickly.

The sequential initial analysis across the whole Wikipedia dataset involves reading the data from disk and than iterating through the whole dataset to perform part-of-speech tagging. Subsequent operations are then based on this data.

6.4.1 Cluster Environment

To evaluate performance it is important to know how the cluster environment was set up. The pipeline ran on 4 nodes with 4 CPUs each (clocked at 2.6 GHz), these are connected via gigabit Ethernet and are running an Ambari installation with YARN and Spark as discussed in the implementation section.

6.4.2 Identifying Nouns

Table 10 shows the performance of the extraction of nouns for the German and English dataset respectively. Keep in mind that this is only the first section of the whole pipeline. Seeing that the this sections runtime should linearly increase with the amount of articles, it should not have a large impact on how the runtime scales. The crossproducts further on in the pipeline leads to quadratic scaling and should therefore have a much larger impact. Combining this with the knowledge that the step of finding nouns initially only takes a fraction of the total computation time the runtime of this step should not be considered as very important for the overall performance.

Table 10 shows that the runtime for finding nouns using part-of-speech tagging is comparable for both languages and, more importantly, scales linearly with the number of articles.

Table 10: Performance of finding nouns in both datasets. English on the left and German on the right.

percentage of articles	runtime
10%	0:13:54
60%	1:15:22
100%	2:10:55

percentage of articles	runtime
10%	0:13:45
60%	0:24:30
100%	0:39:41

This analysis was done on the final pipeline, using Py4J to access the Stanford tagger. Before this tagger was implemented computations with the NLTK based implementation took 11.2 hours for the whole English dataset. Resulting in a speedup of 5 by chaining the tagger implementation from NLTK to the custom one using Py4J. For details on the implementation see Section 5.4.

Computation for a subsection of the articles take proportionally less time. The expected linear growth can roughly be made out from the data, there is a large initial startup time involved, after that the time increase is roughly linear.

For the German dataset using the Stanford part-of-speech tagger "german-fast" with the custom Py4J implementation took 39 minutes. As overall computation takes a lot longer, it is not of great importance for further performance analysis.

Instead, we focus on the potentially bottlenecking operation, the cross product across all articles.

6.4.3 Generating Compound Lists

This subsection makes use of the data gathered until now and combines it with runtime data for the whole pipeline to gain an understanding of the performance of the compound pipeline.

Theoretical Performance Model To understand the best possible performance consider the data flow that is happening. The input dataset is a 12GB Wikipedia File, we perform a few sequential operations on this to turn it into a list of nouns. From the recognizing of nouns section we have a sens of how long this operation takes. For 100% of the data set takes around 40 minutes for the English dataset. This is a good time as it involves loading the complete 12GB file.

It produces around 2.3 million unique nouns. This operation involved in counting will have some performance impact but the huge issue is the total of 5,290,000,000,000 word pairs which need to be evaluated. Even if for each check the CPU only takes a thousand cycles this would mean that the CPUs, which are clocked at 2.6GHz, this operation would still take 200,000 CPU seconds. Due to the overhead of the Python and Java environments this operation would be expected to take much longer. All data that gets sent between nodes has to be deserialized and then serialized again, this operation alone will probably take longer for each pair. Assuming that the checking of each pair takes 4000 cycles instead this still means that all comparisons could be done within 800,000 CPU seconds, seeing as the environment makes use of 16 CPUs with 12 cores each it could enable the check to finish within 12 hours.

Actual Performance Execution time for the whole English Wikipedia dataset was, however, over **200 hours** in practice. While it can be assumed that this is due motley due to the crossproduct let us first take a look at how the performance scales with more data using the German dataset. With

Table 11: These operations were performed on the German Wikipedia dataset.

Percent of articles	time
0.1%	1mins, 46sec
1%	16mins, 41sec
10%	5hrs, 46mins, 34sec

5 hours execution time for a tenth of the whole dataset an acceptable runtime should be achievable if execution times scale linearly. This is however not the case, as we can see the time from 1% to 10% was not just increased by a factor of 10, but instead by a factor of.

Since this pipeline involves use of a crossproduct across all words this would usually result in a quadratic growth of execution times. This growth can be observed, but it should actually be considered in conjunction with the number of unique nouns that were found.

As more articles are analyzed fewer and fewer words are discovered with each article. This results in a decreasing rate of new words being added as articles more are added, this in turn should heavily

impacts execution times. As only unique words are used in the crossproduct in the subsequent section the amount of unique found words has the biggest impact on the resulting performance.

As Figure 9 shows the growth in unique words which are later used in the crossproduct does slow down over time. This means that the runtime of the overall pipeline for finding compound words is expected to have two counter acting effects on it's runtime. On the one hand the crossproduct of found nouns should lead to a quadratic increase in runtime while on the other there should be some counter effect generated by the decreasing amount of new words found.

6.4.4 Word Counting

Word counting is needed for the whole pipeline and presents potentially a large section of the runtime. Comparing different approaches of doing this can therefore help improve performance. In Section 4.2.2, two techniques for word counting were discussed, for simplicity a simple benchmark was implemented separately from either computation pipeline. For this benchmark a tenth of the German Wikipedia dataset was simply split at white spaces and the resulting words were counted. The two approaches outlined in Section 4 were:

1. Count words in a dictionary and for each article and merge the dictionaries using a reduce operation.
2. Create a pair for each word (word, 1) and add them up by reducing by key

As the first approach results in a single dictionary (not an RDD) it is interesting to evaluate the performance impact of transforming the result into an RDD. As the result ultimately is required to be in the from of an RDD for the proposed program.

It seems likely that the approach of emitting pairs is at least a little bit faster. Not only because the second approach does not require the final result to be transformed back into an RDD but also because the final operations in the first approach require working with the entire dataset in a non parallel operation.

Table 12: Execution times compared.

Method	Execution Time
Reduce (with parallelize call)	21min, 47sec
Reduce (without parallelizing)	21min, 38sec
Reduce by key	38sec

When considering Table 12, it is obvious that the method using `reduceByKey` yields much better performance. The additional parallelization is an insignificant factor for the runtime, instead the reduction itself is much slower.

All analysis was therefore done using the second, much fastest approach.

These results are to be expected as both variants of the first approach require some operations to be performed on essentially all data at once. Parallelism can therefore not be exploited as efficiently as in the second approach.

6.4.5 Memory Concerns

As the total amount of data for the English dataset is 12 GB one might not expect there to be memory problems when executing the program on multiple nodes with multiple workers. The overhead of data structures, potentially especially because of the serialization needed for execution with Python. If however memory limits are to low YARN will kill the offending containers, leading to the whole job failing. This means that both the number of workers per node as well as the memory limit per worker need to be chosen sensibly. Ideally each worker node would execute the exact amount of worker processes that uses the total available memory. This way no computation capacity is wasted.

6.4.6 Performance Improvements

In Section 4, a faster approach for forming the tuples of possible compounds was proposed. As the crossproduct is the operation that scales worst (it has a complexity of $O(n^2)$) of those that are used in the pipeline it seems reasonable to look at this first when aiming to improve performance. At some point, no matter the initial execution times, this operation will represent the majority of the execution time.

The performance of the naive approach (a full crossproduct) is compared with the approach that makes use of some additional filtering based solely on the length of the involved words.

Table 13: Avoiding a full crossproduct

Percentage	Duration using:	
	Naive Approach	Simple Length Check
10 %	3:44h	19:56h

Even in very small samples, it is apparent that filtering yields no performance gains and instead has a very negative impact on performance. There is a lot of room to improve performance by other means of speeding up the lookup. This specific method seems to require to much computation for filtering the datasets before the crossproducts. It is possible that there are implementation details that hold this approach back.

6.5 Semantic Relations Performance

The semantic relations pipeline generates essentially a set of vectors for each article that can than be analyzed using any statistics software to perform the clustering. The performance intensive part of this pipeline is additionally finding the IDF for each word. The whole pipeline is relatively simple and offers little room for improving on the macro scale. The group by operations are nothing that can be addressed in terms of performance.

The performance of this pipeline also still leaves things to be desired, currently it runs for **4:22h** analyzing a sample of just 192 words in 10% of the English Wikipedia.

Final analysis was done in scikit-learn as it offers more options than Spark's MLib and has sufficient performance. Clustering of a 200 vectors takes a few seconds on a single machine.

Summary *Results for grouping are promising but hard to assess accurately. It can be concluded that the proposed algorithm has some validity in that it is able to lead to some decent results.*

The results for generating CRA tasks look promising with both techniques and could, with some manual review certainly be used of for actual tasks. The basic building blocks for generating more difficult tasks exist but nothing like this is implemented yet.

The results in trying to solve CRAs vary wildly between the English and the German dataset. Further analysis could show what the exact reasons are, for now we can speculate that it is a combination of the too small dataset and the harder German tasks.

Performance is a hard issue in the Context of Spark that could still be addressed with several different methods.

7 Summary and Conclusion

7.1 Summary

In this work, the full text of both the German and the English Wikipedia were used for two subtasks.

1. Finding Compound Words
2. Finding Semantic Associations of Words

The approach to the first task was to find all nouns in the Wikipedia and evaluate which of those form compounds with any other nouns that were found. PySpark was used to work through the whole Wikipedia dataset and the performance the part-of-speech tagging operation on the whole dataset was good. In this way, a huge list of nouns was created which could then be used to check it for compound words. As this involved checking each noun against every other noun the performance was not acceptable, with the analysis of the whole English Wikipedia taking over 200 hours.

The data generated from the first subtasks was then for the task of both generating and solving CRA tasks. CRA tasks could be generated at a large scale. CRA tasks were solved with an accuracy of up to 33%.

The second subtask was able to cluster words based on their semantics. It was established that this clustering works to some extend and that the vectors representing the words therefor have some legitimacy. The second subtask's results could be used to perform further analysis on how the difficulty of CRA tasks behaves with how words are related to each other.

7.2 Conclusion

Importing the whole Wikipedia dataset seems to provide just enough data to perform the proposed analysis. Larger datasets would be advantageous but without further performance optimizations would result in unacceptable computation times.

Finding semantic association's with the described model appears to work to a degree. Performance does remain an issue with this architecture and existing solutions like Word2Vec yield much better performance, even on single machines. The naive model that was created could be useful in some scenarios. While existing solutions yield both much better performance and results the approach shows that even naive implementations relying on similar assumptions can still work, at least to some degree. For any practical use case one would of course choose to go with an existing implementation.

The generation of CRA tasks works well. The exact quality of the resulting CRA tasks is hard to assess. To guarantee that solutions are unambiguous, a second method for CRA task generation was developed. The advanced method, while slow, also works better for generating unique tasks.

Solving the tasks worked reasonably well for the complete English dataset. 33% of the tasks could be solved. On the German dataset however almost none of the existing CRA tasks could be solved. This could either be due a combination of the dataset being to small (only 10% of the German Wikipedia) and to the German tasks being harder to solve then the English ones.

7.2.1 Technology Choices

While PySpark does offer the great potential benefits with potentially very rapid development pace while still offering good performance there are many issues that make using it a questionable choice.

A lot of the issues during development were only as significant because of the choice of specifically PySpark over the normal Spark version. Static type checking on RDDs can prevent a whole class of bugs where the programmer has the wrong assumptions about the structure of their data. As Spark using Scala does offer type checking for RDDs this could significantly reduce the need for tests and the time needed for debugging.

While PySpark might not be perfect for larger projects but it is certainly great to quickly perform one short tasks. If the task just requires a few operations PySpark can quickly be used. For tasks like sorting and grouping the overhead of using another tool is quickly offset by the speed advantages over pure, single-threaded Python. For this reason some of the post precessing work could be implemented very quickly in PySpark.

7.3 Future Work

Further work on improving the results of both generation and solving of CRAs could involve better filtering of actual compound words. The goal would be to recognize, by comparing some form of word vectors, which words are not a compound but just happen occur in the word. For example the word 'production' has no close relation to the word 'ion' while the word 'ioncannon' is more closely related relation.

There is still a lot of room for improvement of the results. Replacing the whole semantic grouping approach with an established implementation like word2vec would generally improve results and performance.

Further work could also be done in performance enhancements. Seeing that the especially special data structures for looking up words could help cut down the time needed for comparing word pairs to find compounds.

Glossary

common noun A noun used to refer to a class of object or a concept, not a specific name. 34

CRA Compound Remote Associate tasks are used in psychology as a test to gain inside on cognitive processes. The subject is given three words and asked to find a single word that forms a valid compound word with each of these.. 3, 12–14, 17–21, 23, 37–39, 41, 42, 49, 50

DAG A **d**irected **a**cyclic **g**raph is a directed graph that contains no cycles. 6, 31

epenthesis The letters connecting two compound words.. 13

proper noun A noun used to refer to one specific person, object etc. essentially a name. 34

RDD A resilient-distributed-dataset, an abstraction used in Spark to handle parallel operations on data. 5, 6, 16, 25, 47, 49

TF-IDF Value aiming to estimate the significance of a word in a document, given a set of other documents. It is based on the assumptions that words which are frequent words in a given document, are important for a documents content, and words that occur in many documents likely have little to do with their contents . 9, 10, 23

YARN **Y**et **a**nother **r**esource **n**agotiator, a resource manager for scheduling executions of different jobs in a cluster environment. 5, 24, 45, 47

References

[1] Giuseppe Attardi. *Wikiextractor*. URL: https://github.com/attardi/wikiextractor.

[2] Edward M Bowden and Mark Jung-Beeman. "Normative data for 144 compound remote associate problems". In: *Behavior Research Methods, Instruments, & Computers* 35.4 (2003), pp. 634–639.

[3] *Cluster Mode Overview*. URL: http://spark.apache.org/docs/2.0.1/cluster-overview.html.

[4] Evgeniy Gabrilovich and Shaul Markovitch. "Computing semantic relatedness using wikipedia-based explicit semantic analysis." In: *IJcAI*. Vol. 7. 2007, pp. 1606–1611.

[5] *ImageMagick*. URL: http://www.imagemagick.org/.

[6] Trupti M Kodinariya and Prashant R Makwana. "Review on determining number of Cluster in K-Means Clustering". In: *International Journal* 1.6 (2013), pp. 90–95.

[7] Holger Krekel and the pytest-dev team. *pytest fixtures: explicit, modular, scalable pytest documentation*. URL: http://doc.pytest.org/en/latest/fixture.html.

[8] Nina Landmann et al. "Entwicklung von 130 deutschsprachigen Compound Remote Associate (CRA)-Worträtseln zur Untersuchung kreativer Prozesse im deutschen Sprachraum". In: *Psychologische Rundschau* (2014).

[9] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[10] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[11] *pyspark package - PySpark 2.0.1 documentation*. URL: http://spark.apache.org/docs/2.0.1/api/python/pyspark.html#pyspark.RDD.

[12] Paul Schachter and Timothy Shopen. "1 Parts-of-speech systems". In: (1985).

[13] The Internet Society. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. URL: https://tools.ietf.org/html/rfc4180.

[14] University of Stuttgart. *CQPDemon: The STTS tagset*. URL: https://web.archive.org/web/20070911074212/http://www.ims.uni-stuttgart.de/projekte/CQPDemos/Bundestag/help-tagset.html.

[15] *SVG Crowbar*. URL: http://nytimes.github.io/svg-crowbar/.

[16] Kristina Toutanova and Christopher D Manning. "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger". In: *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 13*. Association for Computational Linguistics. 2000, pp. 63–70.

[17] *Wikimedia Downloads*. URL: https://dumps.wikimedia.org/.

[18] *Wikipedia:About*. URL: https://en.wikipedia.org/w/index.php?title=Wikipedia:About\&oldid=735677110.

[19] Matei Zaharia et al. "Spark: cluster computing with working sets." In: *HotCloud* 10 (2010), pp. 10–10.

8 Appendix

8.1 Graphics from Spark Web Interface

Some of the graphics in this document are based on those generated by the Spark Web Interface. Screen shots yield poor quality which is not sufficient for printing, that is why a different approach was used. The graphics in the Spark Web Interface are all SVGs, to extract these from the webpage while retaining the quality and all features of the graphic the SVG Crowbar bookmarklet was used[15]. The resulting SVG file was rendered into a high quality PDF using imagemagic[5].

```
convert -trim -density 1500 in.svg out.pdf
```

8.2 Formalities

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 02.02.2017 Hans Ole Hatzel

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 02.02.2017 Hans Ole Hatzel