SITC2 to ÖNACE mapper

194.047 Interdisciplinary Project in Data Science

Ilir Osmanaj

30-04-2020

Problem Description

In the field of economics, every product and industry category has it's own meaningful name, that should briefly explaing the purpose and with what tha category deals with. Of course, since there are plenty of languages in the world, codes are used to have a world-wide understanding for specific categories.

As of this moment, one very known classification system is SITC (Standard International Trade Classification). SITC is setup and maintained by the United Nations, and in general groups the commodities in a way that they reflect:

- Materials used in production
- Materials used in processing stage
- Market practices and uses of the products
- The importance of the commodities in terms of world trade
- Technological changes

Since the whole industry is continuosly evolving, classification standards should be extended and adapted to the new changes in industry. Currently, SITC is on revision 4 (the versions are named based on revision, e.g. SITC1, SITC2, SITC3 and SITC4).

Other than this international system, since 1970 the EU has developed its own classification system, which is referred as "Statistical classification of economic activities in the European Community", or shortly NACE (from French: Nomenclature statistique des activités économiques dans la Communauté européenne). The NACE system, which basically has the same purpose as SITC, groups economic activites and classifies them with names and codes that are not the same as SITC, but for sure some similarities can be noticed on names.

Austria, as part of EU, has it's corresponding NACE system, named ÖNACE - Österreich (Austrias') NACE. Since not all categories of industries are present in Austria (e.g. production of cotton), ÖNACE has a subsidy of the overall NACE system.

Having multiple classification standards for the economy, sometimes there is the need to convert from one standard to another. In this exact use case, the purpose is to create a mapper, which would map corresponding codes from SITC2 to ÖNACE.

Why is this a hard problem?

In theory, having a mapping system for this shouldn't have been a problem. These categories can be explained briefly in two or three words (and they usually are), but the standards that are analyzed here sometimes use very different naming patters, e.g. Production of beer and wine vs Production of beverages. By meaning, these two categories are *very* similiar, but one cannot say for sure. Imagine the same case, but removing the word production - this way it ends up with 'beer and wine' vs 'beverages'. A human can easily say they are similar, but not a computer - at least not in a straightforward way.

Another thing that makes it even harder, is the length of the descriptions - they are very short (usually 3-4 words). If these descriptions would have been a bit longer, then applying plain Text Search would have been enough. Also, applying more sophisticated information retrieval approaches would be very helpful, too.

Preprocessing steps

Both, SITC2 and ÖNACE files contain more categories than needed for this task. For this reason, some preprocessing was used to cleanup the category description a bit and also select only the items that are needed for the purpose here.

Additionally, the ÖNACE items are used to build the inverted index, which is then used by the TF-IDF weghting explained later.

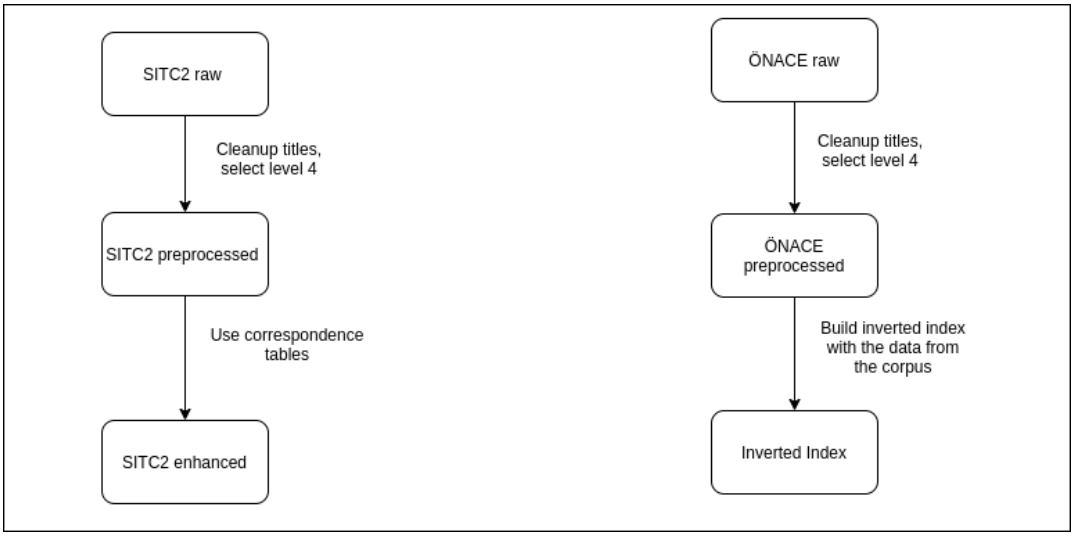

Figure 1: Preprocessing steps for SITC and ÖNACE items

SITC2 preprocessing

SITC2 groups industry categories and commodities in groups and then subgroups. Overall it has ~2580 items, from which 10 are like main groups then there are subgroups for groups, and those subgroups may have groups as well. This can go up different levels of depth, depending on the category. An example for an SITC2 category would be:

- Beverages and Tobacco
  - Beverages
    - Alcoholic beverages
      - Wine of fresh grapes (including must)
      - Beer made from malt (including ale, stout and porter)
    - Non-alcohoic beverages
      - Waters (including spa waters and aerated waters); ice and snow
      - Lemonade, flavoured spa waters and flavoured aerated waters, and other non
  - Tobacco and Tobacco manufactures
    - Tobacco, unmanufactured; tobaco refuse
      - Tobacco, wholly of partly stripped
      - Tobacco refuse
    - Tobacco, manufactured
      - Cigars, cheroots; cigarillos
      - Cigarettes

- Tobacco, manufactured (including smoking and chewing tobacco, snuff); tobacco extract and essences

For the usecase that is being analyzed, only items from category four are selected for further usage. This ends up with 786 items from SITC2 revision.

ÖNACE preprocessing

ÖNACE, same as SITC2, also does groups and subgroups. The same way as for SITC2, only items at level four are selected. For ÖNACE, this means there are only 615 items left for furether mapping.

Data Enhacement

Taking into consideration that this problem of mapping different naming systems should be a fairly common issue, some resources have been used to further enhance the data, so that when corresponding mapping is performed, the algorithms have more data to work with and should therefore perform better on mapping.

A common standard used in economy is the so called Harmonized System (HS). HS is a multipurpose international product nomenclature developed by the World Customs Organization (WCO) and it's used word-wide for customs tariffs and collection of international trade statistics.

There do exist the so called 'correspondence tables', which create mappings between different trading systems. Since HS is widely used, there exist plenty of correspondence tables which map SITC2 classifications to different HS revisions. After having these mappings, those information can be used to extend the SITC2 description and use that text for mapping into ÖNACE system.

For example, what is called "Fish, fresh (live or dead) or chilled (excluding fillets)" in SITC2, is called "Fish; frozen, halibut (Reinhardtius hippoglossoides, Hippoglossus hippoglossus, Hippoglossus stenolepis), excluding fillets, fish meat and edible fish offal" in HS2017. This is for sure a valuable information, since maybe the ÖNACE might contain any of these words and improve the overall search.

HS, as all the other systems, has different revisions. For the data enhacment procedures, the following versions have been used: HS1992, HS1996, HS2002, HS2007, HS2012 and HS2017

Mapping diagram

Even after cleaning up the data and enhancing it with relevant information from the correspondence table, it is very difficult to find a single method that solves the whole problem of mapping.

To overcome this, a kind of 'Ensemble' learning approach was used. The way ensemble learning works is that multiple models get the same input and each of them has a specific output, then either the majority wins, or there is another way of chosing the element - in this case, the intersection of candidates from each approach is chosen as the final list. The best candidate can either be selected automatically, or let the user choose it via a graphical user interface. The approaches used are: text similiarty seach, TF-IDF weighting and Word Embedding.

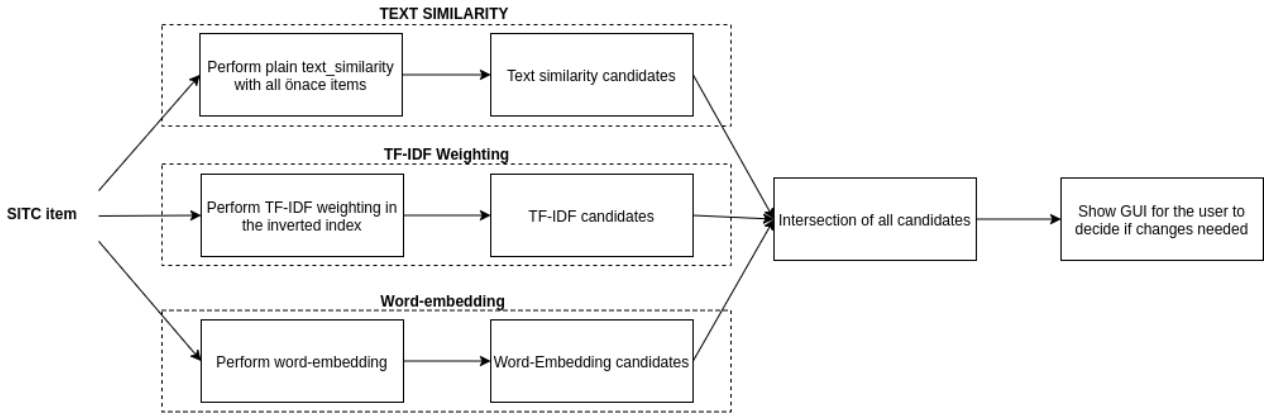

Figure 2: Mapping workflow

Text similarity

In it's core, this can be broken to a pattern search, or just string search. This is a very inuitive thing, therefore just perform plain text search inbetween items and see which items are the most similiar (e.g. have the same words).

The similiarity is performed using the famous Levenshtein Distance Algorithm. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions, or substitutions) required to change one word into the other. Formally, the Levenshtein distance between two strings, a and b (of length |a| and |b| respectively), is given by lev a,b(|a|,|b|) where:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Figure 3: Levenshtein Distance Formula

But, since we have enhanced data, and the text lengths are not of similar lengths any longer, the so called token_set_ratio is used. This means that the whole string is split into tokens, whereas tokens are compared to each other as sets and the result is a ratio of how many tokens mapped with each other. E.g.:

```
>>> token_set_ratio("fuzzy was a bear", "fuzzy fuzzy was a bear")
100
```

As seen, we are interested on the number of tokens that match inbetween two strings and we treat them like sets (meaning that if some token is repeated twice, it will still be counted as one). This decision is based on the fact that there are some items that have the same word repeated more than once, and that does not really add any value on the mapping phase.

The token_set_ration between two strings has a range of 0-100, where 0 means no token match betweent two strings and 100 means all tokens match. For the mapping process used here, a threshold of 70 is used for an item to be considered a reasonable candidate.

In order to choose a reasonable value for the threshold, there has been performed an anlysis to see the impact of the threshold in the following two performance measures:

- Percentage of the mapped items (having at least one candidate)
- Average length of the list of candidates
  - The more candidates, the more unsure the algorithm is

In order to see the impact of data enhacment, both cases are considered in the following charts.

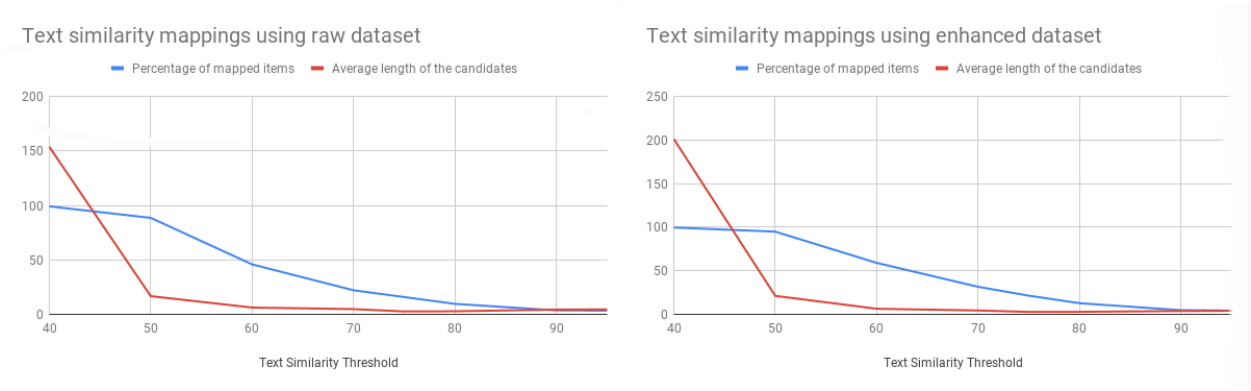

Figure 4: Percentage of mapped items for different values of mapped items

As shown in the visualization, while using a text similiartiy threshold value of 70, about 31% of the items have at least one candidate when using the enhanced dataset, and about 22% of the items have at least one candidate on raw dataset. Other than this number, this threshold makes sense in some manual cases considered by the author as well.

TF-IDF weighting

In information retrieval, TF-IDF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection of documents (so called corpus). It is often used as a weighting factor in searches of information retrieval, text mining, and user modeling. The tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general. TF-IDF is one of the most popular term-weighting schemes today. Variations of the TF-IDF weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query.

Given these properties of the TF-IDF weighting, the mapping of SITC2 to ÖNACE items seems as good use case of it. The list of ÖNACE items is considered the corpus, where we have all the relevant items (each item would be considered a document), and then a search is performed over that corpus, where as user query the SITC2 item description will be used. This should return us all the ÖNACE items relevant to that SITC2 item.

This whole procedure is performed on a data structured called Inverted Index.

Inverted Index

An inverted index is an index data structure storing a mapping from document content (words, numbers - mainly text), to its presence and frequency in a set of documents.

Since python was used for implementing this, dictionary data structure was used to store the inverted index. In general the main processes for building inverted index (excluding word pre-processing) has two components:

- Parsing the documents
  - Reading all the documents (in this case ÖNACE items) and providing dictionaries with önace_code as key, and it's terms as value
- Building actual inverted index

The detailed inverted index creation procedure is as follows:

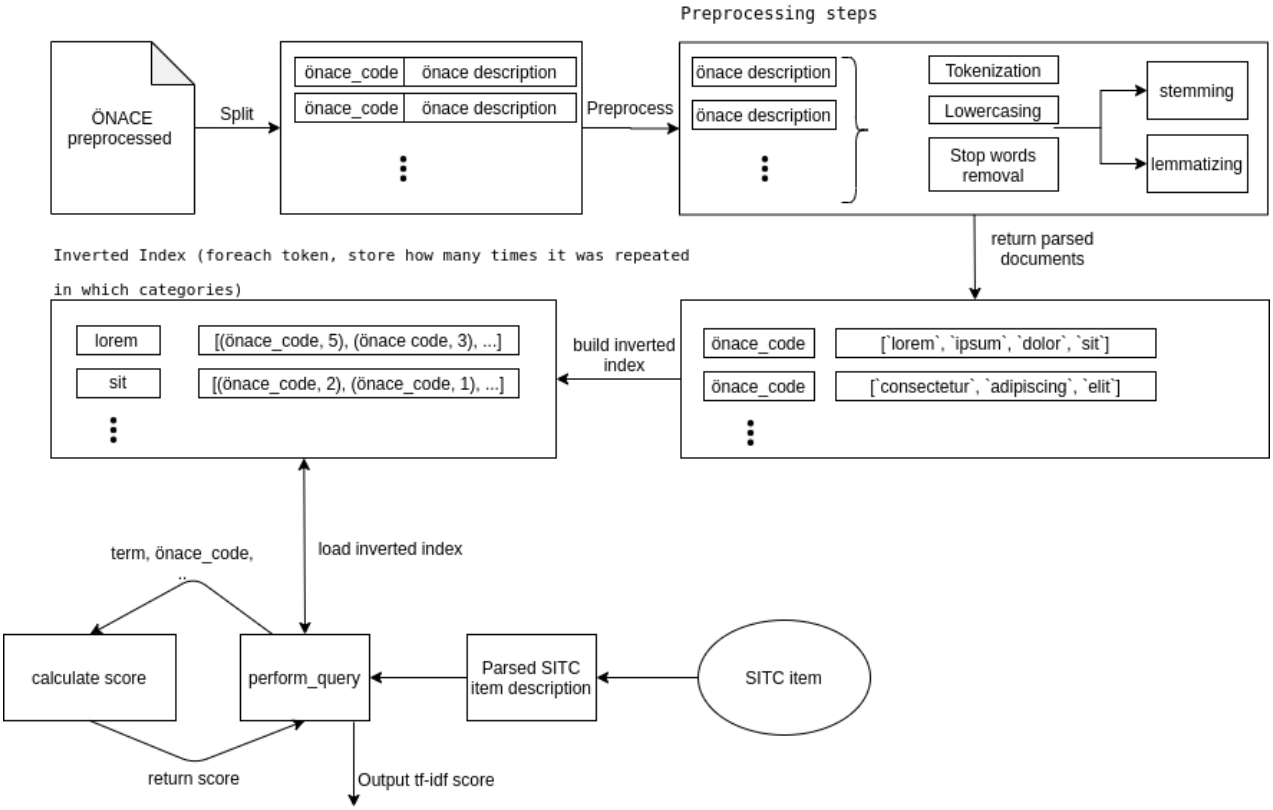

Figure 5: Steps to build the inverted index

Prior to building the actual inverted index, there are some preprocessing steps, which are standard for information retrieval tasks. These include: tokenization, lowercasing, removing of stop words. On top of that, either Stemming or Lemmatization is performed, but never both of them at once.

Tokenization

Given a specific long string, tokenization is the task of chopping it up into pieces, called tokens , perhaps at the same time throwing away certain characters, such as punctuation. These tokens are often loosely referred to as terms or words, but it is sometimes important to make a type/token distinction. A token is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing.

The first step of tokenization, is splitting the whole string by white spaces and then on each token perform cleanup. This clean-ups' purpose is to remove unnecessary characters, e.g. punctuations, quotes etc - for example, we do not want "Austria!" to be different from "Austria". Clean up steps performed on tokens:

- Removal of start and end quotes

- Removal of punctuations
- Removal of brackets
- Removal of double dashes

Lowercasing

A rather simple, but a very important thing in the information retrieval process is lowercasing. It is neccesary, since we do not want "AuStriA" to be different from "austria". Therefore, both the user search query and the document content is lowercased beforehand.

Stop words removal

Consider the words: I, me, you, myself, our, ours, she, yourselves, who, whom etc - these words are called stop-words, and are words that appear very commonly across the documents, therefore loosing their representativeness. These words don't tell much and give no information, therefore it is best to remove them and not use on the inverted index data structure.

This also leaves place for terms which are not so widespread accross documents, but are more representative for them.

Stemming

In the area of Natural Language Processing, we come across the situation where two or more words have a common root. For example, the three words - agreed, agreeing and agreeable have the same root word agree. A search involving any of these words should treat them as the same word which is the root word. So it becomes essential to link all the words into their root word.

Lemmatization

Lemmatization is similar to stemming but it brings context to the words. So it goes a steps further by linking words with similar meaning to one word. For example if a paragraph has words like cars, trains and automobile, then it will link all of them to automobile. For the purpose of mapping here, Lemmatization was used as extra step on top of the basic Information retrieval steps.

Stemming or Lemmatization?

In order to decide which one extra step to perform, a comparison was performed to see which method gives the best mapping results. The results in terms of percentage of mapped items and also the average length of the list of the candidates is given in the following chart.

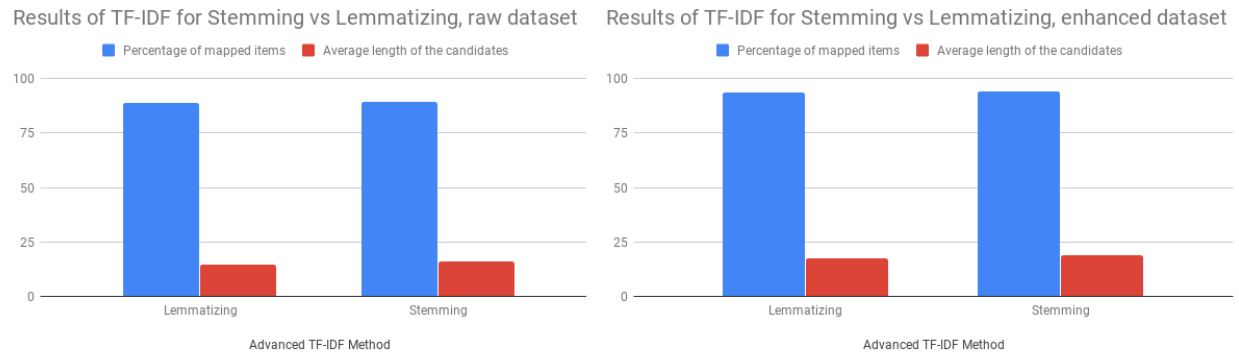

Figure 6: Percentage of mapped items for Stemming vs Lemmatization

As the chart shows, Stemming method performed better in both cases, raw and enhanced dataset, that's why this is used as the default method during TF-IDF weighting.

Scoring

As the name says, TF-IDF is Term Frequency-Inverse Document Frequency approach. Basically, everytime a search is done, that is split into terms and for each terms there should be calculated a weight over all the documents and tell which documents are relevant to the term. For this, two things are important:

- Term Frequency: How often that term is used, in how many documents its shown? If a term is often repeated in a document, it should be seen as highly relevant
- Inverse Document Frequency: if a term is present in a document, but not so present on the others, then that term is highly representative for that document and should increase the weight if present

The general formula:

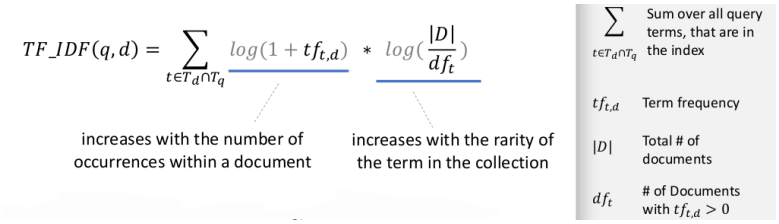

Figure 7: TF-IDF scoring

Word embedding

Word embedding is the collective name for a set of language modeling and feature learning techniques in natural language processing (NLP) where words or phrases from the vocabulary are mapped to vectors of real numbers. Conceptually it involves a mathematical embedding from a space with many dimensions per word to a continuous vector space with a much lower dimension. Methods to generate this mapping include neural networks, dimensionality reduction on the word co-occurrence matrix, probabilistic models, explainable knowledge base method, and explicit representation in terms of the context in which words appear.

Word and phrase embeddings, when used as the underlying input representation, have been shown to boost the performance in NLP tasks such as syntactic parsing and sentiment analysis. As this is an NLP task, this is of great help here to try out.

So, to sum it up shortly, this distributed representation is learned based on the usage of words. and it allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning. Therefore, it goes beyond plain comparison, but takes into consideration the semantics as well. When two different words are compared against each other, the vector representation of these two words is retrieved and their distance compared. There are two distance metrics usually used: cosine similiarty (which is very specific to vectors) and word movers distance.

There are different options one can choose in this situation: build it's own word embedded model, or use the already pre-trained ones. Even though the data available have been enhanced here, there is still very few data and training a model on these available data would lead to a biased and bad model. Pre-trained models, on the other hand, are trained on plenty of data from different sources and are therefore much better for these kind of NLP tasks. Also, they are trained for you, so it saves time and resources.

For the purpose of this task, the word2vec embedding is used.

Word2Vec

Word2vec is a two-layer neural net that processes text by vectorizing words. Its input is a text corpus and its output is a set of vectors: feature vectors that represent words in that corpus. While Word2vec is not a deep neural network, it turns text into a numerical form that deep neural networks can understand. The purpose and usefulness of Word2vec is to group the vectors of similar words together in vectorspace. That is, it detects similarities mathematically. Word2vec creates vectors that are distributed numerical representations of word features, features such as the context of individual words. It does so without human intervention.

Given enough data, usage and contexts, Word2vec can make highly accurate guesses about a word's meaning based on past appearances. Those guesses can be used to establish a word's association with other words (e.g. "man" is to "boy" what "woman" is to "girl"), or cluster documents and classify them by topic. Those clusters can form the basis of search, sentiment analysis and recommendations in diverse fields.

The output of the Word2vec is a vocabulary in which each item has a vector attached to it, which can be fed into a deep-learning net or simply queried to detect relationships between words.

An example of Word2Vec for the word 'Sweeden', would output in order of proximity:

- Norway, 0.760
- Denmark, 0.715
- Finland, 0,620
- Switzerland, 0.588

As seen, this approximity is learned by the words usages in similar contexts - and it's not that bad.

While comparing two sentences, each of the words is compared in terms of distance, and for this use case the Word Movers Distance has been used.

Word Movers Distance

Other than the cosine similarity, word movers distance is now considered among state of the art metrics for text comparison. From the high level overview, it uses word embeddings to learn semantically meaningful representations of the words, based on local co-occurrences in sentences.

WMD (Word Movers Distance) utilizes the property of word vector embeddings and treats text documents as a weighted point cloud of embedded words. The distance between two text documents A and B is calculated by the minimum cumulative distance that words from the text document A needs to travel to match exactly the point cloud of text document B.

Consider the case of two sentences:

- Sentence 1: Obama speaks to the media in Illinios
- Sentence 2: The president greets the press in Chicago

The first step, as in all text retrieval task, is removing the stop words and getting the vector representations for each word. Then, two documents are compared against each other word by word, as in the following image.

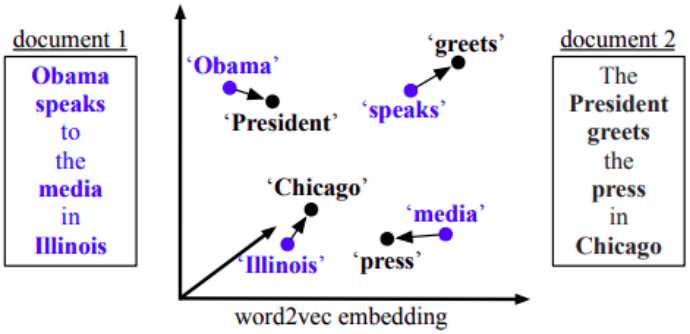

Figure 8: Word Movers Distance example

Since sentences have multiple words, and choosing which word to use for pair-waise comparison can lead to multiple comparison. Therefore, the two most similar words are choosen to be compared (e.g. Obama with President, Press with Media, Chicago with Illinois etc). The smaller the distance, the more similar the sentences are to each other.

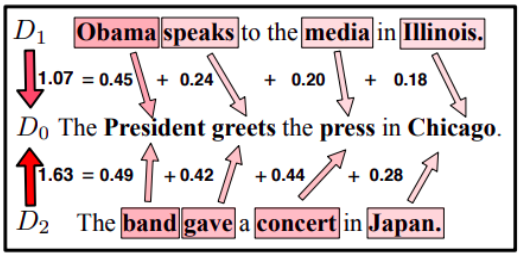

Figure 9: Word Movers Distance example (cont.)

In the end, the similarity of two sentences using WMD is just a number. For the problem of mapping SITC2 items, a threshold of what is considered similar would be needed. But, since we are not sure whether there actually exist similar items that reach a certain threshold, we compare the distance with all OENACE items, and then choose top 10 candidates with the smallest distance, as possible mapping candidate. Of course, for the final result we use intersection of all candidates from all approaches, that's why we can be safe this will not introduce any bias in the output.

Mapping Results

TODO: write something about the results. How many items are mapped? why so few? how long it takes etc. Show examples when we find mapping, show that sometimes we fail because of no intersections etc

Graphical User Interface

Even though there are different approaches trying to solve this issue in the background, this is a sensitive process which needs some human having a look at it - at least for a bit. For this purpose, a GUI application was built, which to the user the posibility to list the list of intersecting candidates for each SITC2 item, select one of them and even delete or change it.

This is a simple GUI, which offers the basic CRUD (create, read, update, delete) operations on the current mappings. Also, user is offered the possibility to load existing mappings and start working on them from there. The GUI tool was built mainly for convinience of the user.

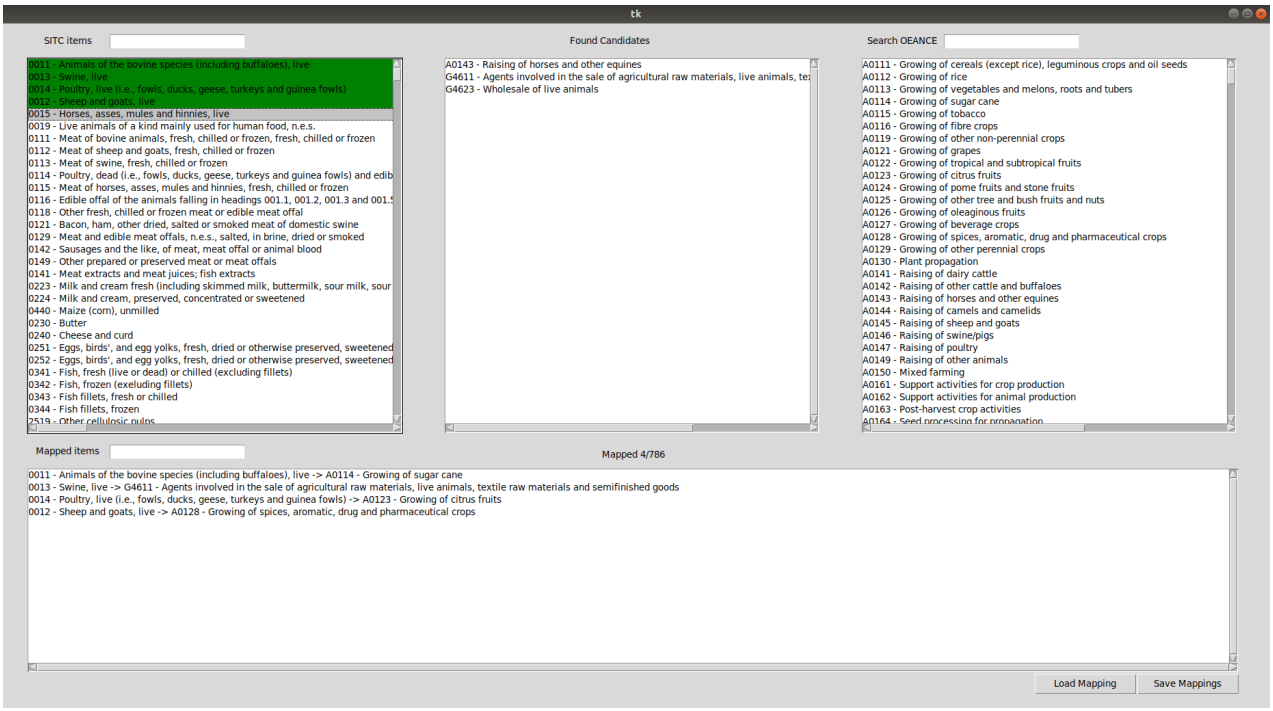

Figure 10: Graphical User Interface

All in all, there is a list of SITC2 items in the left. When an item is clicked, it shows the found candidates and the list of ÖNACE items is always there, to manually search and map it.

Conclusions