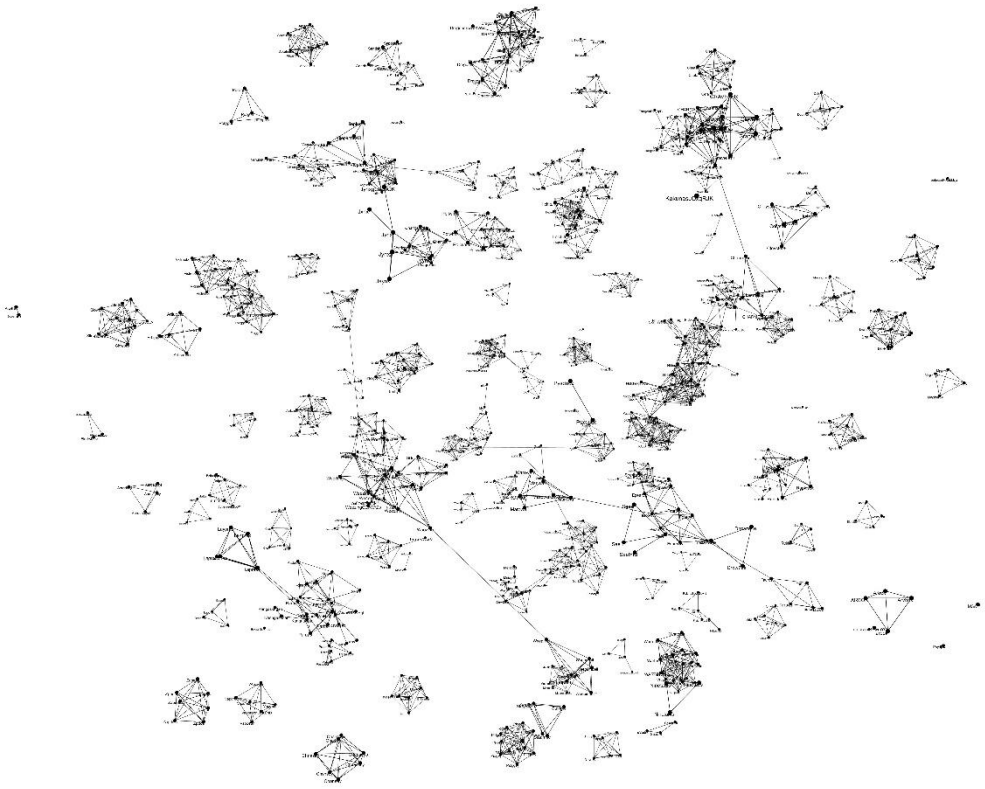


Text Field Clustering to improve Data Quality for a successful Migration

Author: Georg Vetter

Contact: georg.vetter.privat@gmail.com

1. Project Overview

The aim of the project is to improve data quality, particularly with regard to duplicates or inaccurate duplicates. Duplicates are objects in a database that represent the same thing but are written differently. If the duplicates are written identically, they can be removed from the database relatively easily. However, if they are written differently, this becomes more difficult. Therefore, the goal of this project is to find out which objects are actually the same even though they are written differently. This project belongs to the domain of data quality improvement. The idea for the project was derived from a work project in a migration project in which data had to be migrated from a legacy system to a new system. It was important that the data quality in the new system was high. Specifically, this involved a case in which texts could be freely entered in a column in the old system, while only a selection of characteristics was possible in the new system. In order to insert the old data into these selection options, it was necessary to find out which strings belonged together and which represented the same thing in order to enable an assignment. As the data is internal company data, it unfortunately cannot be transferred or submitted for security reasons. For this reason, I have decided to develop a data generator that generates the same real-world objects in different notations. This generated data should then be clustered.

2. Problem Statement

2.1 Problem Definition

There are several columns in a database that contain strings. These columns are filled by people with free text. This involves information that does not necessarily require a free text field. For example, device types or manufacturers of devices are entered as free text. This leads to unnecessary heterogeneity and diversity in the strings. The strings that are entered as free text are characterised by possible typing errors, as the entry is made either via an app or a keyboard. It is not checked whether the input is valid or how good it is. This is a problem in terms of data quality.

It becomes problematic when machines have to interpret these strings. As part of a migration project, these strings should be merged so that all real objects have the same name or string. This is the only way to migrate these objects, as the new system has restrictive conditions for accepting data from previous systems.

The target system requires that the strings in this column can no longer be entered as free text fields. Instead, only a selection of predefined values may be entered. In future, the person on site entering the string will have a list of selection options and select the appropriate value, which is then transferred to the column. This will ensure and improve data quality in future compared to the current system. However, we have to transfer our old data to the new system in order to be able to view historical data. The challenge here is that we have different characteristics than those envisaged in the target system. We therefore have to transform our data in the old system so that it fits into the new system.

Another problem is that the target system does not yet know exactly which characteristics are required and which should be offered. This means that we cannot simply assign the old data to the existing clusters or characteristics. Instead, we have to think for ourselves about which real objects we have in our legacy data and which need to be mapped in the new system. For this reason, we cluster the legacy data by summarising similar real-world objects with different spellings. The correction of various spelling mistakes and typos is the task of this project.

2.2 Solution strategy

My rough strategy and approach to solving the problem was to first simulate training data, as I did not want to use internal company data. Then I wanted to look for other papers published on the Internet that might have solved this problem and how they went about it. Basically, I expect a definition of a similarity measures, which describes the similarity of two words and which would be used to cluster the words. Then I would look to describe an algorithm that clusters the training data in such a way that it maximises a previously defined quality measure. At the end, I want to output a visualisation in which I can see how well these clusters are defined. To set the algorithm, I will probably use a grid search algorithm to set the best hyperparameters.

The following outline shows the procedure again in a structured way. I will then provide further details on each individual point.

1. Generation of Training Data
2. Similarity Measures
3. Model Construction
4. Hyperparameter Optimization
5. Training and Testing
6. Limitations
7. Potential Extensions

3. Solution Approach

3.1 Generation of Training Data

As it was clear that I didn't want to use internal company data for this project, I had to think about training data or sample data that had a similar structure and the same problem as the company data, but had no connection to it.

That's why I went looking for generators for spelling mistakes and typos. There are various generators on the Internet, which I looked at and found very useful. However, I decided to create my own generator, simply in order to take the programming experience with me and orientated myself strongly on the generators found on the Internet.

The generators from the Internet use several categories of spelling and typing errors that they have added to given words. In the following, I will briefly describe which changes and word transformations I have incorporated as spelling and typing errors in my generator. The code for the generator can be found in the project in the file TypoGenerator.py.

3.1.1 Different capitalisation

The simplest typo is the different capitalisation of words. This means that the first letter can be written in upper or lower case, but the second letter can also be written in upper case. For each word, a variant is created with upper case first letters, lower case first letters, upper case first two letters and lower case first two letters.

Example:

word
Word
WOrd

3.1.2 Additional characters when pressing the Enter key

When entering words in an input mask and confirming the entry with an Enter key, the person making the entry may hit another key in addition to the ENTER key. On the German keyboard, this is the + or the # which is then often found at the end of a string.

Example:

Word+
Word#

3.1.3 Omitting blanks

Spaces are also frequently omitted, especially in the case the company's internal data. In the case of internal company data, for example, this also involved type names with spaces. There, for example, letters and number combinations are separated by a space, where the omission of the space represents a typing error.

Example:

Word 123
Word123

3.1.4 Swap letters

When typing, it can happen that you swap two letters in a word so that two neighbouring letters appear in the string in the wrong order. All letter combinations in a string were swapped and then output as variants from the generator.

Example:

Word
oWrd
Wrod
Wodr

3.1.5 Duplicate letters

In addition to swapping letters, double pressing of a letter is also common. For example, all letters in a string have been doubled in position.

Example:

WWord
Woord
Worrd
Wordd

3.1.6 Missing letter

In the rush of typing, it can also happen that you forget letters and therefore missing letters appear in a string. Here, too, I have omitted letters in some positions to simulate this typing error.

Example:

ord
Wrd
Wod
Wor

3.1.7 Neighbouring button instead of actual button

A somewhat more complex typing error is replacing a letter with a neighbouring letter on the keyboard. To do this, I looked at the German keyboard layout and defined a neighbour for each letter on the keyboard. In the generator, each letter was then replaced position by position with all its neighbours.

Example:

Qord
Eord
Sord
…

3.1.8 Neighbouring button in addition to the actual button

Of course, a neighbour can not only replace a typed letter, but can also appear before or after a letter. So it can happen that when I press the K, L, J, M, I or O appear before or after this K.

Example:

Weord
Wqord
Wsord
…

3.1.9 Noise at the end of the word

Finally, I added another misspelling source, which could also be found in the company's internal data. Some people added further information to the name, which had nothing to do with the name and was not recurring. I simulated the whole thing by adding a noise factor at the end of each word, which was a random character string.

Example:

Wordlsdijf
WordAfüjF
Wordpf
…

3.1.10 Using the typo generator

Now I had a typo generator, but still no basic data on which this typo generator could work. I needed a string data set that had the same properties as the company's internal data set. Basically, the internal company data set can only have a limited number of characteristics, but these can be written differently. So, I want to have a fixed set of strings whose spelling I can vary using the generator. To give the project a bit of a fun factor, I decided to use all Pokémon names of the first generation as input and to vary these Pokémon names using the typo generator and then to be able to cluster them again later.

The following table shows an excerpt of the generated training data.

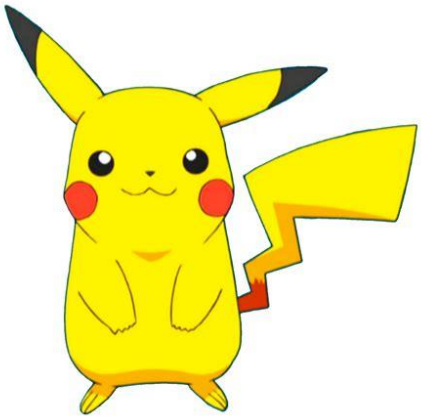
	word	cluster
0	Alakaza	Alakazam
1	Vulpix#	Vulpix
2	Arcanne	Arcanine
3	Syndslash	Sandslash
4	Rhyhron	Rhyhorn
5	Pigey	Pidgey
6	Rhyhorn#	Rhyhorn
7	poliwrath	Poliwrath
8	Nidoarnm	Nidoranm
9	Rjyhorn	Rhyhorn
10	Wigglytuff	Wigglytuff
11	Hypno	Hypno
12	Clefable	Clefable
13	Carmeleon	Charmeleon
14	Gaastly	Gastly
15	Gengarr	Gengar
16	Victreebel	Victreebel
17	Sanddslash	Sandslash
18	PErsian	Persian
19	Zapdos	Zapdos
20	Kabutops	Kabutops
21	Pnsir	Pinsir
22	Parasdct	Parasect
23	Koffing	Koffing
24	Pinsir#	Pinsir
25	Shellder#	Shellder
26	HitmonchanDLmBVp	Hitmonchan
27	Koffing#	Koffing
28	Gro2lithe	Growlithe
29	Mukyo	Muk
30	Butterfree	Butterfree
31	Krabby+	Krabby
32	Chsnsey	Chansey
33	DragonitelAwv	Dragonite
34	Primeapee	Primeape

Figure 1: Example of Typos from the Typo-Generator

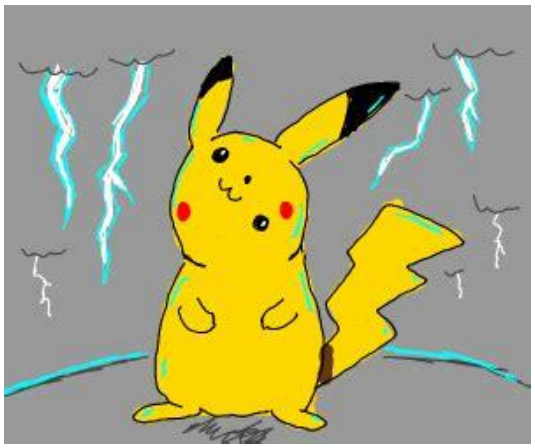
The word column contains the varied variant of the word. The original name is in the cluster column. Some words have not been varied at all, so that the same string is in word and cluster, e.g. line 10. The aim is therefore to summarise all the different spellings in the word column that have the same characteristics in the cluster column, as in the following illustration.

	word	cluster
56	Pikachu#	Pikachu
137	Pkiachu	Pikachu
197	PikachuoXiJrknH	Pikachu
292	Pikahcu	Pikachu
446	PikachuoXiJrknH	Pikachu
530	Pikachu#	Pikachu
534	Pikachujq	Pikachu
766	pikachu	Pikachu
939	PikachuhXL	Pikachu

Figure 2: All Variants of a Cluster



Pikachu



Pkiachu

A total of 151 Pokemon names were entered into the generator, which created a total of 1360 variations.

3.1.11 Expected results

Since the initial data is relatively well balanced and no cluster is excessively large and the bootstrap is drawn randomly, I assume that the clusters determined by my algorithm are approximately the same size.

3.1.12 Output of the training data

The generated variations were not output in full, as smaller or larger amounts of data should also be tested for the algorithm, a bootstrap is drawn (with a return) and this is then output. The size of the bootstrap is freely selectable.

3.2 Similarity measures

You can find various similarity measures for strings on the Internet. One is already known from Udacity; the cosine similarity. In addition to cosine similarity, I also found fuzz ratio (based on the Levenshtein distance) and Jaccard similarity and considered them relevant for the project. I also defined my own similarity measure: Sub String Similarity. All four similarity measures are briefly described below.

3.2.1 Cosine Similarity

Cosine similarity is a measure used to quantify the similarity between two vectors in n-dimensional space. It is often used in information retrieval, text mining and machine learning. The formula for calculating the cosine similarity between two vectors A and B is:

$$Cosine\ Similarity(A, B) = \frac{A \cdot B}{\|A\|\|B\|}$$

$A \cdot B$ stands for the scalar product of the two vectors and$\|A\|$ and $\|B\|$ for their respective lengths or norms. The cosine similarity measures the cosine of the angle between the vectors and lies between -1 and 1, with a higher number meaning a greater similarity between the vectors. If the Cosine Similarity is close to 1, the vectors are similar, whereas if they are very different, they are close to -1. If the cosine similarity is close to 0, the vectors are orthogonal to each other.

3.2.2 Fuzz Ratio

The fuzzy ratio is a measure of similarity between two strings based on the Levenshtein algorithm to calculate the number of changes required to convert one string to the other. The formula for calculating the fuzzy ratio is:

$$Fuzz\ Ratio = \frac{2 \times Matches}{Length\ of\ String1 + Length\ of\ String2} \times 100$$

Here, "Matches" stands for the number of matching characters in the two strings, and "Length of String1" and "Length of String2" are the lengths of the strings. The fuzzy ratio is between 0 and 100, where 100 means a perfect match. The higher the fuzzy ratio, the more similar the strings are. This metric is often used in text processing and information retrieval to assess the similarity between texts, especially when it comes to typing errors or spelling variations.

3.2.3 Jaccard Similarity

Jaccard similarity is a measure of the similarity between two sets. In the context of texts, the sets are considered to be the set of unique words in each text. The Jaccard similarity $J(A, B)$ between two texts A and B is defined as the ratio of the number of common words to the total number of different words in both texts:

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

Here $|A \cap B|$ is the number of words that occur in both texts, and $|A \cup B|$ is the number of different words in both texts together. The Jaccard similarity is between 0 and 1, where 1 means that the two texts are identical and 0 means that there are no words in common. Jaccard similarity is often used in information retrieval, text analysis and machine learning methods for comparing texts and for cluster analysis.

3.2.4 Sub String Similarity

The "Sub String Similarity" method checks whether a word occurs within another word and evaluates the similarity between these two words. If a word is completely contained in another word, the Sub String Similarity is defined as 1 to indicate that they are identical or very similar. Otherwise, the similarity is scored as 0 to indicate that they do not overlap in any way or have common parts. Mathematically, the sub-string similarity between two words A and B can be defined as follows:

$$Sub\ String\ Similarity(A,B) = \begin{cases} 1 & wenn\ A\ ein\ Substring\ von\ B\ ist\ oder\ umgekehrt \\ 0 & sonst. \end{cases}$$

3.3 Model Construction

3.3.1 Ensemble-Clustering

Instead of choosing a single similarity measure, I decided to use all similarity measures. Thus, for each word combination, each similarity measure should be calculated and in the end, calculated into a similarity score, which is calculated from the sum of all similarity measures, with a respective weight.

$$Similarity\ Score(word1, word2) = Cos * w_{Cos} + Fuzz * w_{Fuzz} + Jaccard * w_{Jaccard} + Sub * w_{Sub}$$

The advantage of this approach is its flexibility. Individual similarity measures can be excluded by setting their weight to 0, but other measures can also be emphasised by setting their weight to the top.

3.3.2 Clustering with Threshold

After the similarity score has been calculated for each word combination, a decision must now be made as to the similarity score above which two words should be combined to form a cluster.

$$Connection(word1, word2) = \begin{cases} 1 & if\ SimilarityScore_{word1,word2} > Threshold \\ 0 & else \end{cases}$$

3.3.3 First Model

To test the procedure, I initially set the parameters to a reasonable level for me. All similarity measures are weighted equally ($w_{Cos} = w_{Fuzz} = w_{Jaccard} = w_{Sub} = 0.25$) and a threshold of 0.5 was selected. As an initial assessment of the goods, I then visualised these clusters and assessed the quality myself.

3.3.4 Visualisation

I used two visualisations to determine the quality of my clustering by hand. One visualisation was two-dimensional, in which the words were assigned to clusters in a graph. The second visualisation was three-dimensional in order to get a better overview and to be able to adopt different perspectives. Both visualisations are based on the Python package networkx, which I use to build a graph that is to be visualised later. The two-dimensional visualisation was created with Matplotlib and looks like this.

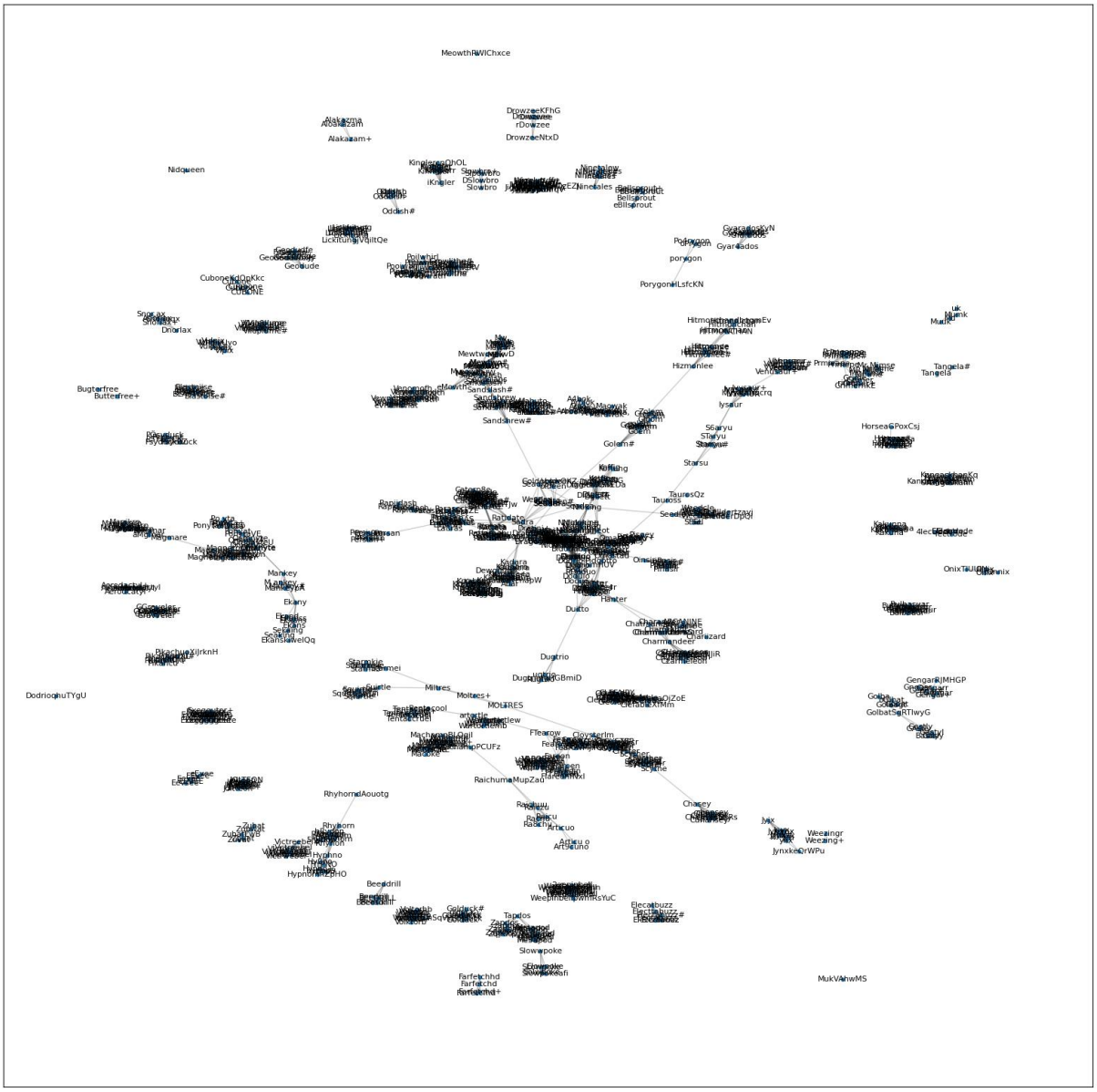


Figure 3: 2D Visualisation of the First Model

Here we can see that this static two-dimensional visualisation does not show which words belong to which cluster. Only an overall impression can be conveyed. You can see that there are many smaller clusters on the outside, which would correspond to my expectations. However, the large cluster in the centre does not. All Pokemon names should appear roughly equally often, which is why the clusters should all be roughly the same size. This is a first indicator that the selected initial parameters are not optimal. However, we can still look at sections of this visualisation in detail.

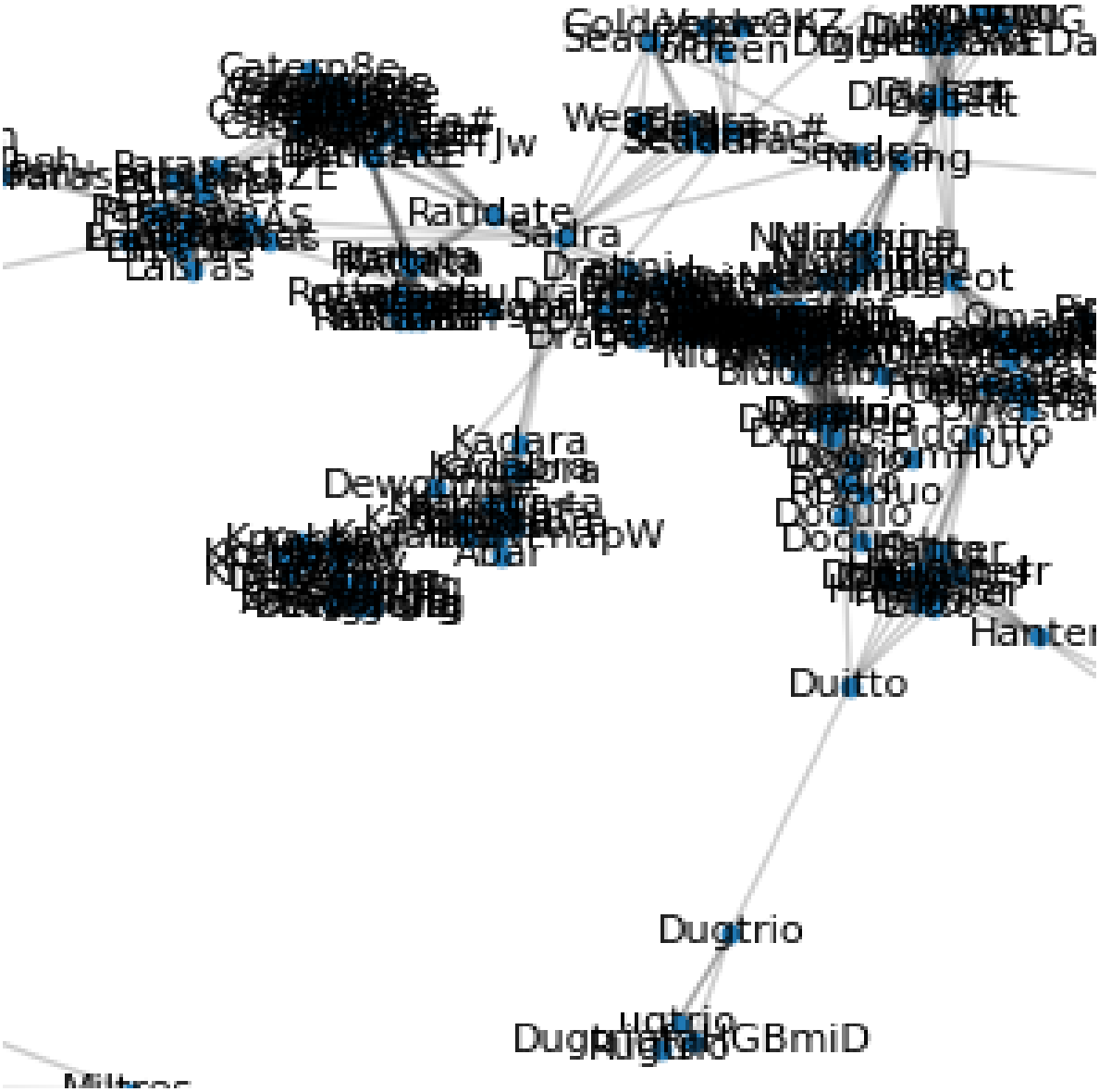


Figure 4: Zoom of the 2D Visualisation of the First Model

Here it becomes clear that reading the individual words is difficult even on closer inspection. However, you can also see here that words have been clustered that do not appear to belong together.

Due to the close proximity of different words and the overlapping labels. I found it appropriate to use a three-dimensional visualisation in addition, in which I can view the clusters from different perspectives and thus better visualise the overlaps. The following figure shows the visualisation in 3 dimensions using the Gravis package in Python.

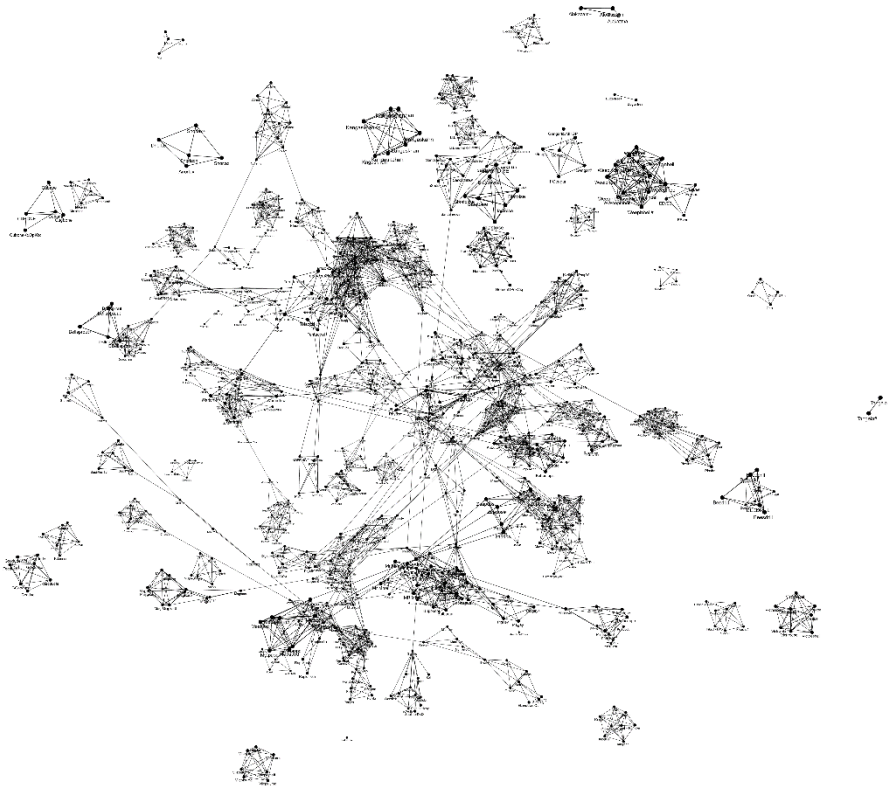


Figure 5: 3D Visualisation of the First Model

As this visualisation can be rendered in HTML, it is possible to fly through the clusters with the mouse and take a closer look at individual clusters. To make it easier to understand the visualisation, it is attached as an html file (Initial_Parameters.html).

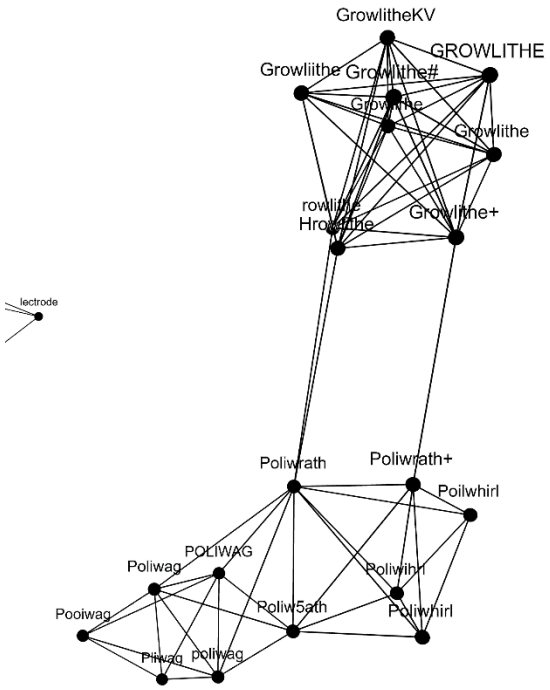


Figure 6: Zoom of the 3D Visualisation of the First Model

Here I have selected an example cluster in which it is clear that the threshold of the initial parameters still appears to be too small, as a relatively large distance between two clusters was bridged here and both clusters, which refer to different objects, were merged into one cluster.

It is important to note that the position of the words in two- and three-dimensional space does not correspond to what is expressed by the similarity score. All unassigned words are freely distributed in the space. And thus do not represent the proximity to existing clusters. However, the length of the edges corresponds to the similarity score, which simplifies the assessment of the threshold.

The F1-Score of this first model is about 0.84. As you can easily see from the visualisationand the F1-Score, the initial parameter settings are not optimal. For this reason, we wanted to optimise these hyperparameters.

3.4 Hyperparameter Optimisation

There are various hyperparameters in my cluster algorithm that need to be optimised. These include the weights for calculating the central similarity score, but also the threshold, which determines the level of similarity at which two words are combined into a cluster.

3.4.1 Appropriate quality measures

In order to assess whether one hyperparameter combination is better than another, a suitable quality measure must be found. There are various classic quality measures to choose from, such as precision, recall, accuracy, F1 score, but also the basic metrics such as true positives, false positives, false negatives and true negatives.

In order to be able to assess the quality measures, it must first be shown how this quality is to be calculated. Two matrices are shown below for this purpose. The first matrix shows the connections between two words that were determined using the algorithm (adjacency_matrix). The second matrix shows the actual connections between two words that were given in an input (y).

adjacency_matrix	TentacoolBYwmYQ	Tentacool#	Tentacruel#	Tentacruel
TentacoolBYwmYQ	1	0	0	0
Tentacool#	0	1	1	0
Tentacruel#	0	1	1	1
Tentacruel	0	0	1	1

y	TentacoolBYwmYQ	Tentacool#	Tentacruel#	Tentacruel
TentacoolBYwmYQ	1	1	0	0
Tentacool#	1	1	0	0
Tentacruel#	0	0	1	1
Tentacruel	0	0	1	0

The diagonal entries are negligible as they only indicate that the word is related to itself. Due to the strong imbalance in the matrices (sparse matrix), most quality measures are not applicable.

The basic metrics such as True Positives, False Positives, False Negatives and True Negatives are unfortunately unsuitable to act as quality measures on their own. Each of these metrics only represents a part of the necessary truth that I need to determine a corresponding quality.

	True	False
Positive	1	1
Negative	3	1

Increasing **true positives** and reducing **false negatives** could be achieved by associating all words with all words. Increasing **true negatives** and decreasing **false positives** can be achieved by not making any connections.

Since the mapping matrix is a sparse matrix, the **accuracy** cannot be used either. The large number of true negatives achieved when no connections are made far outweighs the few cases where true positives would have a positive effect.

If I allow all connections, I have a good **recall**. Therefore, the recall is also not suitable. **Precision** is a very good quality measure for the highly imbalanced data, but the **F1 score** can also deal well with this imbalance. In my project, I therefore decided to use the F1 score.

3.4.2 Grid Search

The parameters mentioned above were varied in the grid search. Both the threshold and all weights could take on values between 0 and 1, whereby the grid was used in 0.05 steps. The weights are interdependent, which is why only permissible weight combinations were tested here in which the sum of the weights equals 1. The course of the mean F1 score across all weight combinations per threshold is shown below.

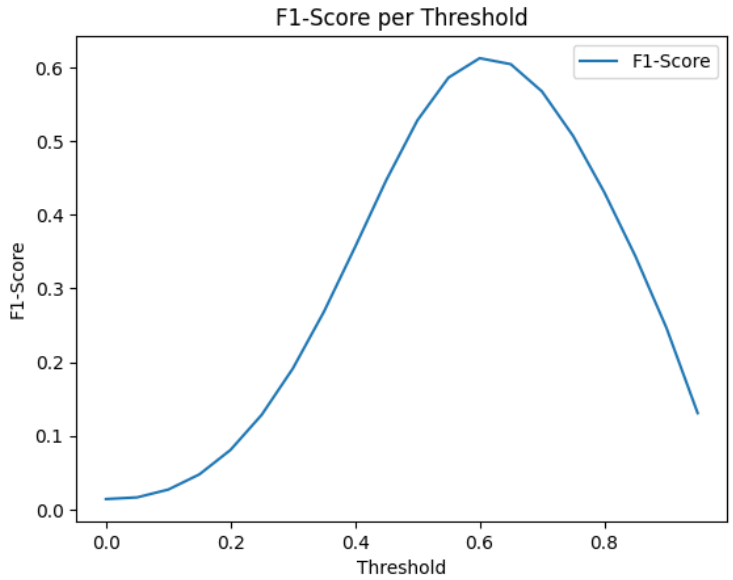


Figure 7: Mean F1 Score Curve across all Weight Combinations for each possible Threshold

The best parameter combination consists of a threshold of 0.65 (the graph only shows the mean value of the F1 score across all weight combinations per threshold, which is why the highest point is not necessarily 0.65), and the weights $w_{Coss} = 0.05$, $w_{Fuzz} = 0.65$, $w_{Sub} = 0.1$ und $w_{Jaccard} = 0.2$. This parameter combination achieved an **F1 score of over 0.90**. This means that all similarity measures are relevant for clustering, but fuzz similarity is the most important for success. The threshold shows that no similarity measure alone can be responsible for the connection between two words. Even if a similarity measure is set to 1, no connection is formed between the words if the other measures are set to 0. This means that several measures are required to reach the threshold.

If we look at the two visualisations, the following picture emerges.

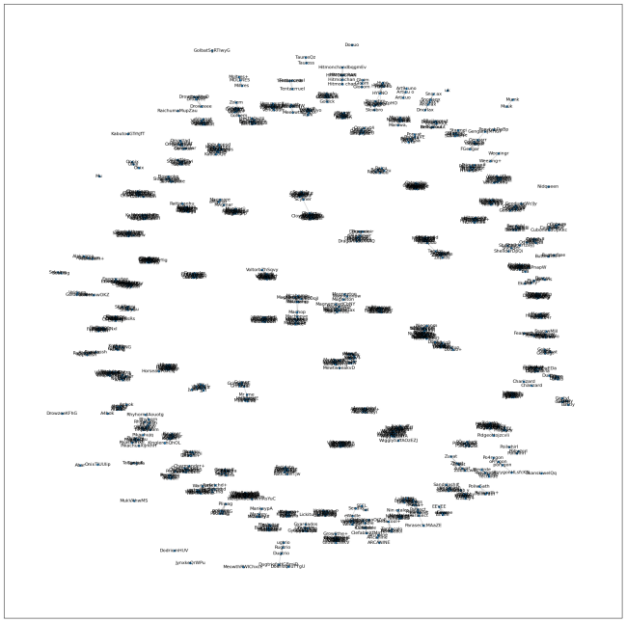


Figure 8: 2D Visualisation of the optimised Parameters

We can see that there are many smaller clusters and no single large cluster. If we zoom in on a sub-area, we can at least roughly assess the quality of the clusters further.

Figure 9: Zoom of the 2D Visualisation of the Optimised Parameter Model

Even though many of the words overlap, you can see that the clusters look quite good. To check this impression further, let's look at the three-dimensional image.

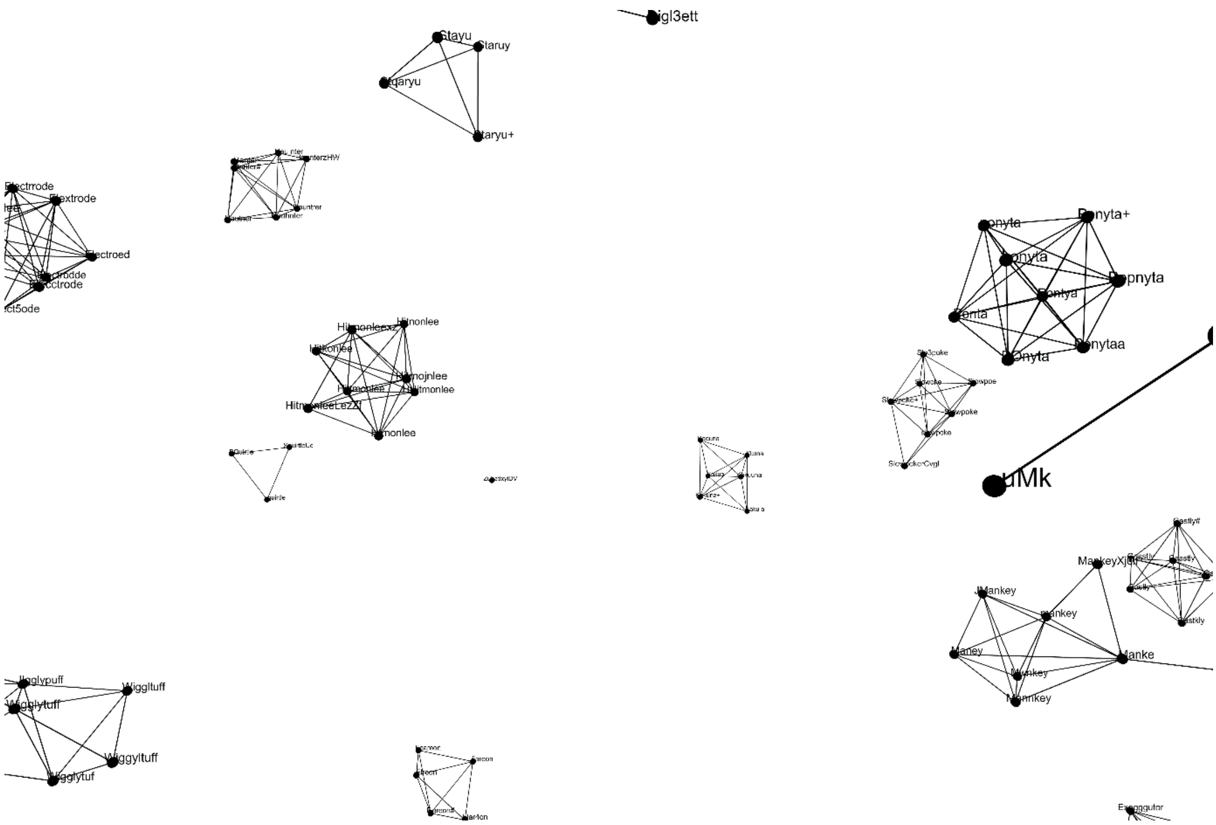


Figure 10: 3D Visualisation of the Model with optimised Parameters

Here, too, the clusters generally look very good, although there are still inaccuracies, such as the exemplary zoom in the next image.

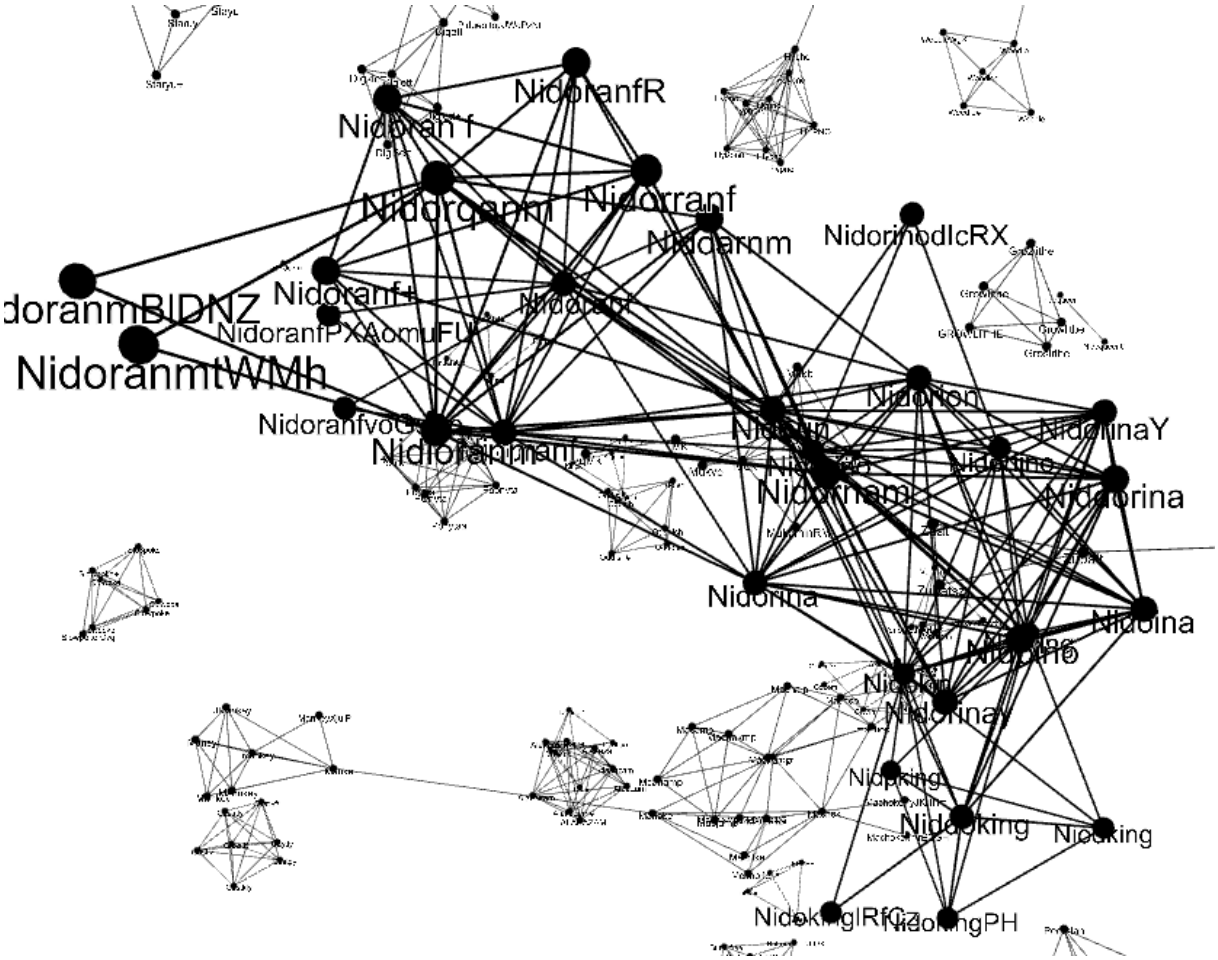


Figure 11: Zoom of the 3D Visualisation of the Model with optimised Parameters

Here you can see that the Pokémon, whose names are very similar by nature, have also been merged into clusters. This error will be difficult to solve algorithmically and will probably have to be corrected manually. To make it easier to understand the visualisation, it is attached as an html file (Best_Parameters.html).

3.5 Training und Test

As a small subset of the output data is clustered at most in the real use case, the procedure will be tested below with a split of training and test data. Here, the input data is split into a training set and a test set using train_test_split. The grid search algorithm is again used for the training set and then the test set is clustered using the determined weights.

The two-dimensional visualisation already shows that the algorithm works well. If we look at the F1 score, it is also above 0.90. This method also works for the separation of training and test data, at least for this synthetic data.

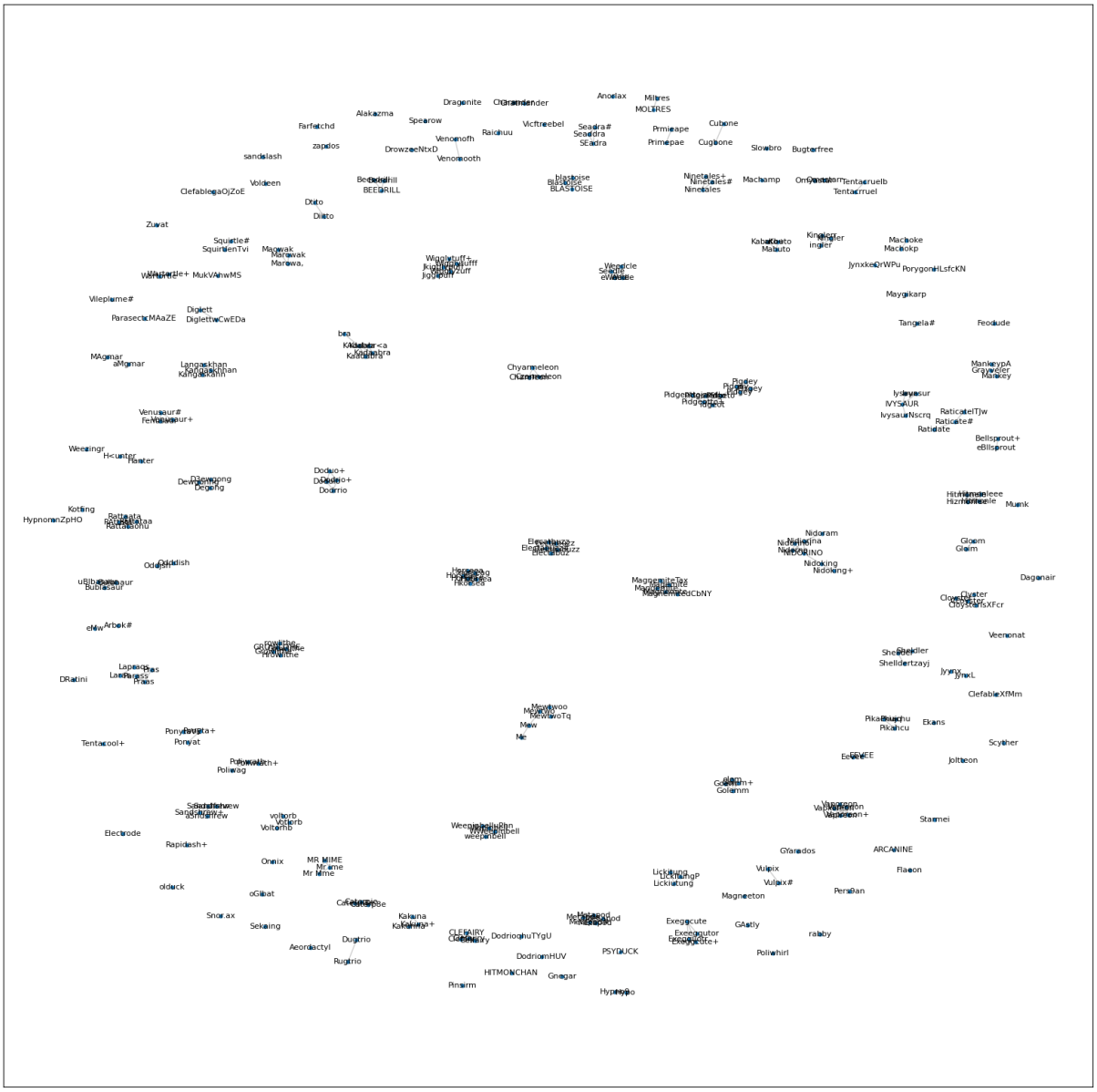


Figure 12: 2D Visualisiation of the Test Set

To make it easier to understand the 3D visualisation, it is attached as an html file (Test_Visualisation.html).

4. Conclusion

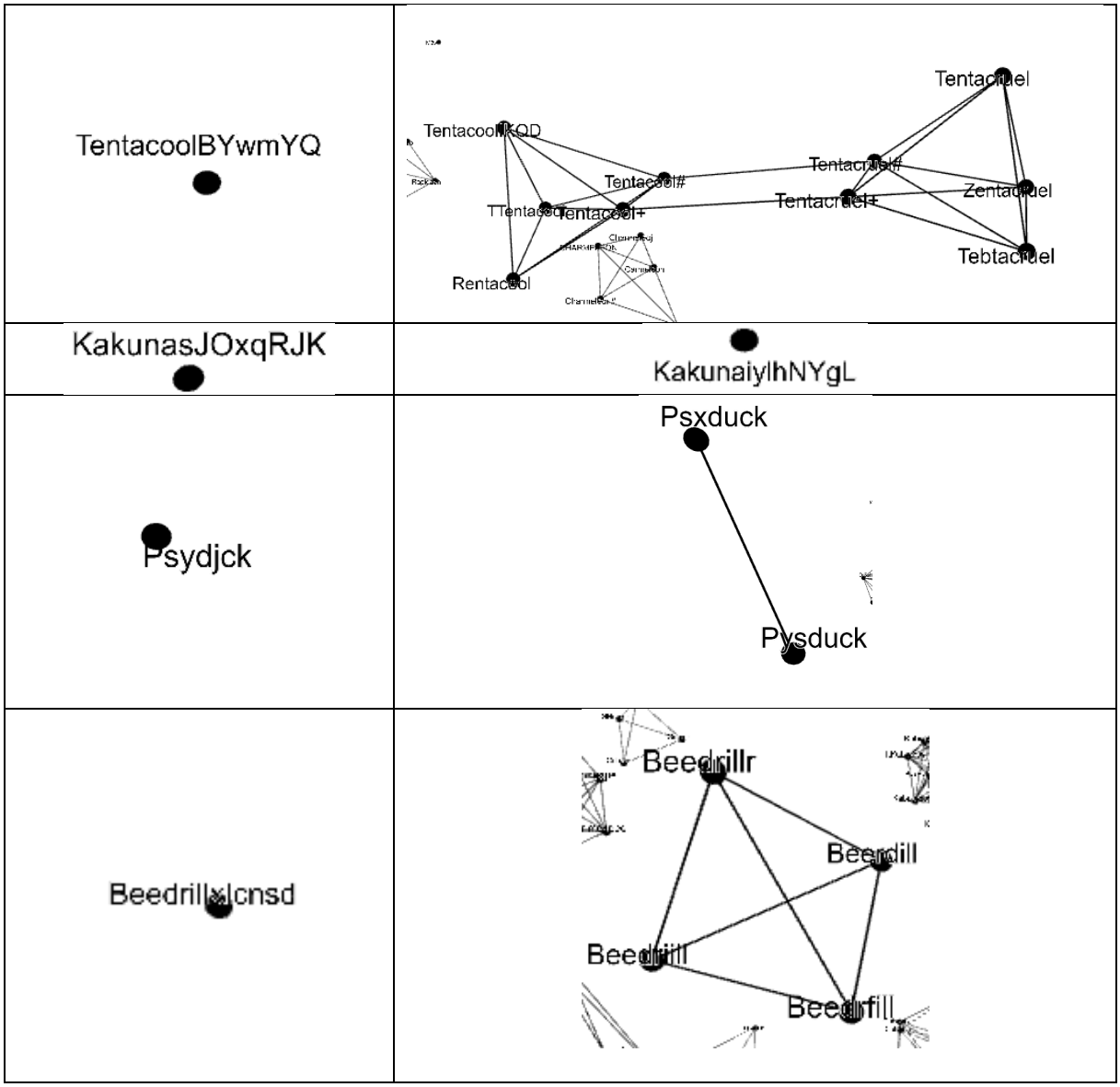
An algorithm was built that forms clusters for the synthetic data. When generating the data, an attempt was made to create a data set that was as realistic as possible. This data could then be formed into clusters using an initial model. A grid search was then applied to all the data to set a benchmark for the quality of the clusters. Finally, the method was used when the weights are only trained on a subset and a test set is to be clustered with these weights. The results have shown that the method works well on the generated data.The first model achieved an F1 score of 0.84, the optimised model an F1 score of 0.909 and the model with separate training and test sets an F1 score of 0.902. This shows that the procedure was successful.

What I found particularly interesting about the project was the creation of the data set, as this gave me a deeper insight into the characteristics of spelling mistakes. I also found it exciting to investigate and

apply the various similarity measures. Even though there is a lot of overlap between the measures, the project shows that each measure contributes to the clustering.

4.1 Limitations

The data analysed is synthetic data, which is why its applicability must be re-examined in each individual case. Even after the grid search, the weights and the threshold are not set so optimally that no errors occur in the training data. After the grid search on the training set, I noticed the following errors in the visualisation. Words that belong to the clusters were not included in the clusters, but also in the case of Tentacool and Tentacruel, clusters were merged that should actually be separate.



How these weaknesses of the algorithm could be cured is considered in the final chapter.

4.2 Potential Extensions

Further measures to improve the quality of the algorithm include, for example, the addition of distance measures such as Euclidean distance or Manhattan distance. However, this results in a greatly enlarged

parameter space that must be searched in the grid search. Therefore, other optimisation methods, such as an evolutionary strategy, can be used to find the optimal weights.

Further attributes per word can also be added. These can be, for example, additional information on the real-world objects or word vectors that create a separate word for each word, as is already implemented in the NLTK package.

Finally, machine learning methods can be applied to the extended set of attributes and thus achieve increased quality if necessary.