University of Bergen
Department of informatics

Analysis of Word Embeddings: A Clustering and Topological Approach

Author: Jonas Folkvord Triki

Supervisor: Nello Blaser

UNIVERSITETET I BERGEN

Det matematisk-naturvitenskapelige fakultet

June, 2021

Abstract

Over the last few years, advances in natural language processing (NLP) have enabled us to learn more from textual data. To this end, word embedding models learn vectorized representations of words by training on big sets of texts (e.g. the entire Wikipedia corpus). Word2vec is a word embedding model which learns single vector representations of words. However, by creating such single vector representations of words, it becomes hard to separate between word meanings, as the single vector representations have to cover all the word meanings. Words with multiple meanings are called *polysemous*, and determining the word meanings is a challenging problem in NLP. Traditionally, word embeddings from word2vec are analyzed using analogy and cluster analysis. In analogy analysis of word embeddings, it is common to show relationships between words, e.g. that the relationship between *king* and *man* is the same as that between *queen* and *woman*, whereas, in cluster analysis of word embeddings, it is common to show how similar words cluster together, e.g. the clustering of country-related words. Moreover, due to recent developments in the field of topological data analysis, a topological measure of polysemy was introduced, which attempts to identify polysemous words from their word embeddings. The goal of this thesis is to show how word embeddings traditionally are analyzed using analogies and clustering algorithms and to use methods such as topological polysemy for identifying polysemous words of various word embeddings. Our results show that we are effectively able to cluster word embeddings into groups of varying sizes. Results also revealed that the measure of topological polysemy was inconsistent across word embeddings, and our proposed supervised models attempt to overcome and improve on this work.

Acknowledgements

I would like to thank all the people who helped and supported me with the work of this thesis.

Firstly, I would like to thank my supervisor, Nello Blaser, for his outstanding guidance during the work of this thesis. Your expertise and insight have brought the quality of this work to a higher level, and I am very grateful for that.

Secondly, I would like to thank the research group in machine learning at the Department of Informatics at UiB for providing me with computational resources. The computational resources have helped me performing the analyses in this thesis, which I would not have been able to at an equal scale otherwise.

Thirdly, I would like to thank my fellow graduate students, particularly Naphat, for the long lunch breaks and the helpful academic discussions.

Lastly, I would like to thank my family and friends for all their love and support. Specifically, I would like to express my gratitude towards my father and Nora. You have always been there for me and have helped me to rest my mind outside the thesis.

Jonas Folkvord Triki
01 June, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Natural language processing (NLP) is a field of study which focuses on the interactions between computers and the natural human language [Allen, 2003]. In modern NLP applications, it is common to see the usage of vector embedding algorithms, such as the word2vec techniques introduced in [Mikolov et al., 2013a]. Word2vec learns vectorized representations of words (called *word embeddings*) by training on big text sets (e.g. whole Wikipedia). Word embeddings are created such that the semantics of a word is reflected by its word embedding. For example, the word *solution* is similar to the word *chemistry*, but is also related to words such as *answer* or *equation*. By inspecting the word embeddings of such words, we find that they are highly similar. Furthermore, we also see that the word *solution* can have multiple meanings, depending on the context it is used in. We call such words *polysemous*, and the task of determining polysemous words in NLP is a difficult problem. To measure and determine polysemous words from word embeddings, the measure of *topological polysemy* was introduced in [Jakubowski et al., 2020]. The topological polysemy method seeks to identify singular word embeddings, which the authors claim reflect polysemy.

Classically, word embeddings are analyzed using analogy and cluster analysis [Mikolov et al., 2013a, Mikolov et al., 2013b, Walkowiak and Gniewkowski, 2019]. In this thesis, we will first perform a similar analogy and cluster analysis, by using general methods from machine learning, and among these, clustering algorithms and dimensionality reduction methods. The goal is to find a richer description of word embeddings. In particular, we will first explain how we trained and evaluated our word2vec model, before proceeding onto clustering of word embeddings using clustering algorithms. We will validate the results from the cluster analysis using internal cluster validation methods and visualize our results using dimensionality reduction methods. After performing the cluster analysis on word embeddings, we will in this thesis investigate the method of topological polysemy by applying it to various word embedding models. Our goal is to recreate the results shown in [Jakubowski et al., 2020] for a general word embedding model, which further strengthens their proposed method and results. We will also look at another algorithm called *Geometric Anomaly detection* (GAD). GAD seeks to identify singular points in the data, similar to what the topological polysemy method does. Following, we will compare the results using topological polysemy and GAD to show their relation. Next, we will look at the application of intrinsic dimension estimation (ID) methods on word embeddings and show the relationship between the estimated local ID of a word to the number of word meanings. We would like to, in particular, see if the estimated local ID of word embeddings can help us to predict the number of word meanings. Finally, we propose two supervised methods for predicting polysemous words, using results from topological polysemy, GAD and ID estimation methods.

This thesis is structured as follows. In Chapter 2 we give the technical and theoretical background required for this thesis. Following, in Chapter 3 we perform our analysis of word embeddings, explaining training and evaluation steps for our word2vec model, perform clustering of word embeddings, and at last, analyze methods for polysemous words prediction. In Chapter 4, we summarize and conclude the thesis. Finally, in Chapter 5 we look at ideas for future work related to the thesis.

Chapter 2

Background

In this chapter, we will introduce and explain the technical and theoretical background of this thesis. In particular, we will introduce general methods from machine learning, the notion of word embeddings, and topological data analysis (TDA) and methods from its field. We will describe concepts to the point where they are understandable for the sake of the thesis, but leave out technical details of concepts that are not in focus (e.g. intrinsic dimension estimation in Section 2.1.6). On the other hand, we have some particular concepts which we have worked with more extensively throughout the thesis, such as clustering (Sections 2.1.2 and 2.1.3), word embeddings (Section 2.2), and TDA (Section 2.3). Thus, we will give a thorough explanation of these concepts. Finally, we assume that the reader is familiar with general concepts from calculus, linear algebra, and statistics. In the following sections, we will introduce and explain the technical and theoretical background of this thesis.

2.1 General machine learning methods

In this section, we will introduce and explain general machine learning methods. In particular, we will first define machine learning in Section 2.1.1, then move onto clustering algorithms and validation of clusters in Sections 2.1.2 and 2.1.3. Following, we will describe two methods for performing dimensionality reduction in Section 2.1.4, explain the concept of artificial neural networks in Section 2.1.5 and intrinsic dimension estimation in

Section 2.1.6. Finally, we round the section by explaining methods from regression analysis in Section 2.1.7, how to perform model selection in Section 2.1.8 and introduce performance metrics in Section 2.1.9.

2.1.1 What is machine learning?

In traditional computer programs, we typically create the rules and instructions of the program to get the output we desire. In *machine learning* (ML), however, we turn the problem on its head, and the goal is to learn the rules of the program using its input data [Mitchell, 1997]. More formally, we quote the famous definition from [Mitchell, 1997, p. 2] to define ML: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E". To give an example, let E be data of whether or not it has rained the last seven days in Bergen, and let T be the task of determining if it rains the next day. To measure the performance P, we use the percentage of correct guesses for whether or not it rains the next day. The ML program can then use the data from experience E to improve on the task T by maximizing the performance P. Furthermore, we motivate the use of machine learning by illustrating with an example in Figure 2.1, where we see the differences between traditional programming and machine learning programs.

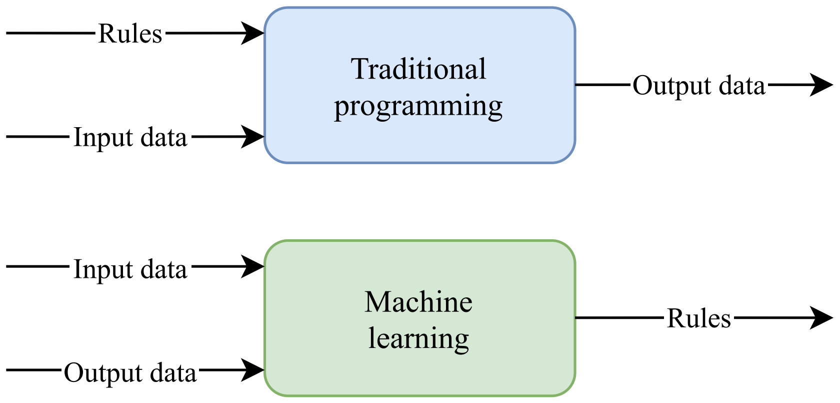

Figure 2.1: Traditional programming compared to machine learning.

In this thesis, we will in particular look at two approaches for ML, namely *supervised* and *unsupervised learning*. In a supervised learning setting, we give the ML program its input and output data. The goal is to learn the parameters of the ML program to map the input data to the output data. In an unsupervised learning setting, however, we do not give the ML program any output data, leaving the ML program to find structure in its input data. The goal of unsupervised ML programs is to discover and learn hidden patterns from the input data, by applying (possibly) several methods. In this thesis, we will learn from data by mainly using unsupervised ML methods (e.g. cluster analysis and dimensionality reduction in Section 3.2), but also some supervised ML methods (e.g. supervised polysemy prediction in Section 3.3.4) as well. In the following subsections, we will introduce and explain general methods and concepts from machine learning.

2.1.2 Clustering algorithms

In a supervised machine learning setting, we typically use data X and its associated labels y. The supervised task is to train a model to predict the labels y using the data X. A classical example of a supervised machine learning task is to distinguish between dogs and cats, where X is an image of a dog or a cat and the labels y indicate whether or not the data X represents a dog ($y = 0$) or a cat ($y = 1$). In an unsupervised setting, however, the labels y are less likely to be present. To predict the labels y, we apply *clustering algorithms*.

Clustering is one of the most important methods in unsupervised machine learning. Clustering algorithms divide some data X into clusters (groups) such that the data in each cluster are similar in some sense. An example of this is clustering by using *Euclidean distance*, which measures the distance of a line segment between two points u and v. More formally, we define the Euclidean distance between two points u and v as

$$d(u, v) = ||u - v||_2 = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + \ldots + (u_n - v_n)^2}, \tag{2.1}$$

where u and v are two n-dimensional vectors. If we cluster by Euclidean distance, we want the distance between any two data points belonging to the same cluster to be small. We refer to this distance as the *intracluster distance* or *compactness*. Unfortunately, it is usually not enough to only minimize the intracluster distance; we also have to ensure that the distance between the clusters is as large as possible. To measure the distance between two clusters we measure the distance between two data points belonging to different clusters. We refer to this distance as the *intercluster distance* or *separation*. If a clustering algorithm can create clusters such that we have small intracluster distance and large intercluster distance, it indicates that the clustering algorithm is good for the data at hand. We note, however, that data can be complex and it can be hard to find good clusters. In the following sub-subsections, we look at some common clustering algorithms, explain how they work, and discuss their strengths/weaknesses. In each of the clustering algorithms, we assume that we have some data $X = \{x_1, x_2, \ldots, x_n\} \in \mathbb{R}^{n \times d}$. Furthermore, we will use the clustering algorithms below in the analysis of word embeddings (Section 2.2) in Chapter 3.

2.1.2.1 k-means clustering

The *k-means clustering* method is an unsupervised machine learning algorithm for identifying clusters in data [Bishop, 2006, Section 9.1]. The algorithm uses an iterative approach to search for k clusters, where k is a *hyperparameter* (i.e. in control by user). There exist several variants of this algorithm and we discuss two of them in later sub-subsections (see Section 2.1.2.2 and Section 2.1.2.3). Following, we explain the standard (and naïve) variant of the algorithm (i.e. *Lloyd's algorithm*), and we base this sub-subsection on [Bishop, 2006, Section 9.1].

The standard k-means clustering algorithm works as follows. The first step is to determine the initial cluster means, or *centroids*. Since we want the algorithm to output k clusters, we have to decide k initial centroids. The simplest way to do this is to select k random data points to be the initial k centroids. The next step is to calculate the Euclidean distance between each data point to the cluster centroids. We do so because we want to determine which cluster each data point belongs to. Furthermore, we assign each data point to its closest cluster centroid and compute the mean of each cluster. The third and final step is to move the cluster centroid to the new mean of each cluster. We repeat the second and third steps until convergence is met (e.g. change of loss is less than a set threshold). Finally, we illustrate the use of k-means clustering in Figure 2.2, where we see its cluster boundaries, representing the clusters. The white crosses represent the cluster centroids.

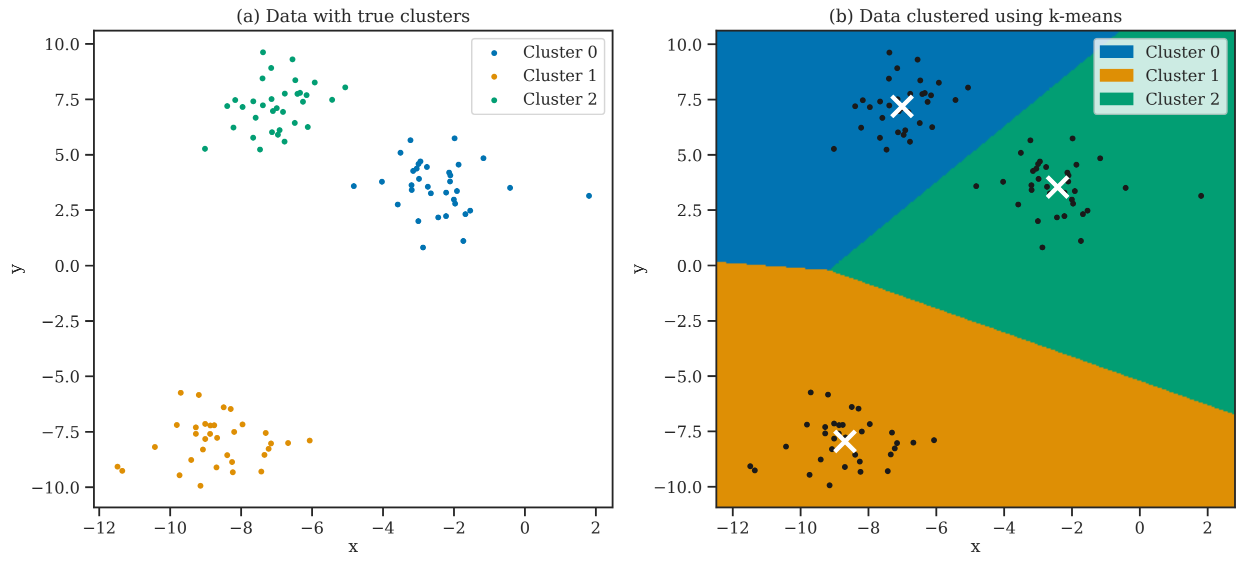

Figure 2.2: Example of clustering using k-means clustering on a 2-dimensional data set with 3 clusters. We show the cluster boundaries in (b), emphasized by the colours. The white crosses represent the cluster centroids.

Mathematically, the goal of k-means clustering is to minimize the squared distance between the points in each cluster to its respective centroid, which we refer to as the within-cluster sum of squares (WCSS). The objective is to find

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2, \tag{2.2}$$

where $C = \{C_1, C_2, \ldots, C_k\}$ are the clusters of the data X, k is a hyperparameter for the number of clusters and μ_i is the cluster centroid of cluster C_i.

The main advantage of k-means clustering is its simplicity, both in implementation and when interpreting the results. The algorithm also scales well to larger data sets, and there is only one hyperparameter to tune (number of clusters k). As for the disadvantages, the algorithm is rather sensitive to the initialization of centroids in the first step. If we were to select bad initial centroids, the convergence time of the algorithm increases greatly, and we might end up with bad clusters. We also have to choose the number of clusters manually, which is a downside if we have no additional knowledge of the data beforehand. In addition to this, the algorithm suffers from the *curse of dimensionality*, which is a set of problems that can occur when analyzing high-dimensional data (i.e. greater than two to three dimensions). In the context of k-means clustering, as the dimensionality increases, it becomes more and more difficult to distinguish between data points (all points converge to the same distance). We illustrate an effect of the curse of dimensionality in Figure 2.3.

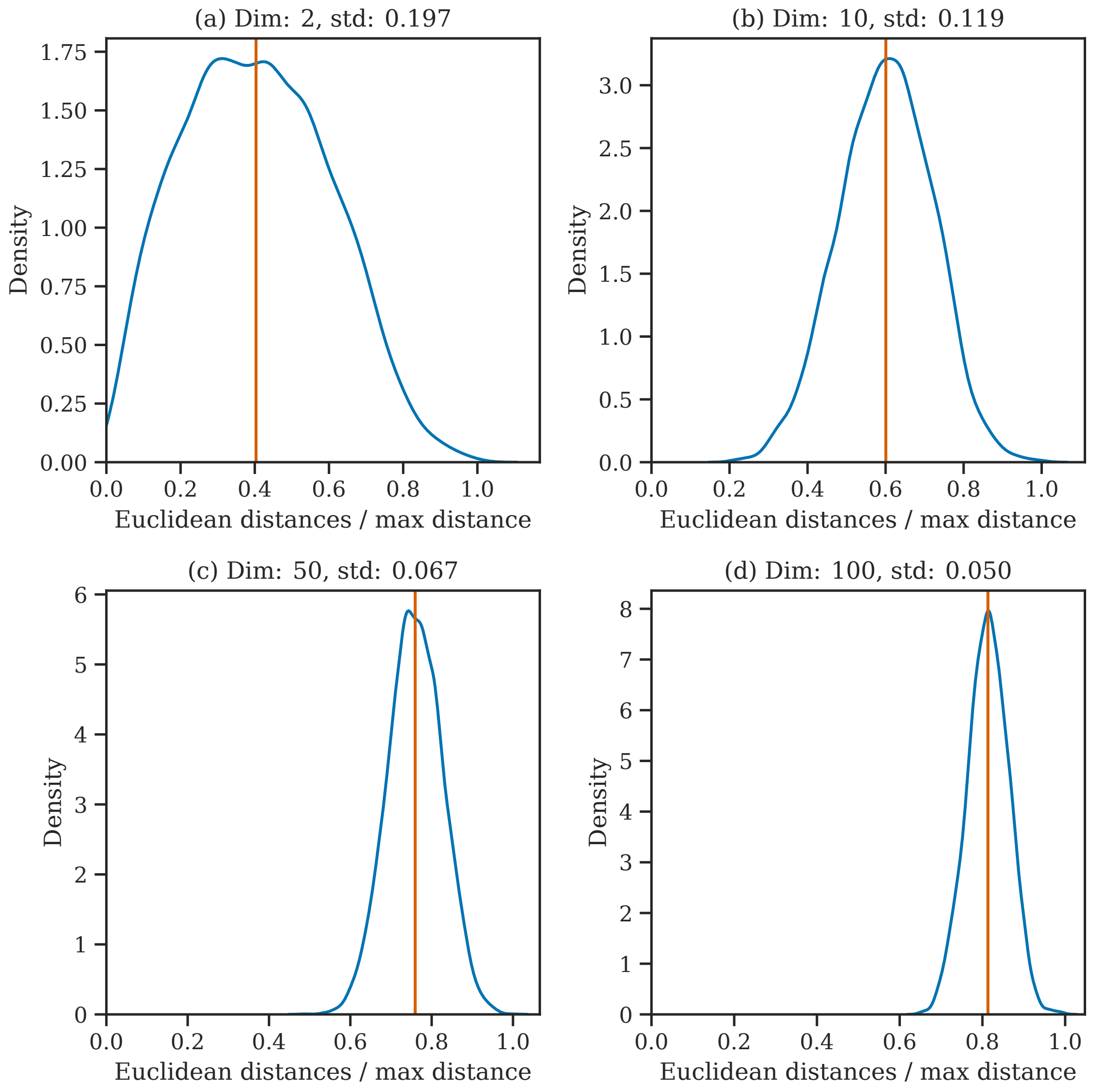

Figure 2.3: Four illustration plots showing an effect of the curse of dimensionality. The distance between points in higher dimensional spaces becomes the same as the dimensionality increases, and thus, it is harder to differentiate between points using distance metrics. The blue line represents the density of the pairwise Euclidean distances (we divide by the max distance to normalize the x-axis). The red line is the mean distance.

2.1.2.2 Mini-batch k-means clustering

Mini-batch k-means clustering is a variant of k-means clustering, where we use *mini-batches* of data points to find a suitable clustering [Sculley, 2010]. To create the mini-batches, we randomly sample subsets of the training data for each training iteration (similar to the mini-batches in gradient descent from Section 2.1.5.6). We refer to [Sculley, 2010] when explaining mini-batch k-means clustering.

The algorithm is similar to the standard k-means clustering algorithm and comprises two major steps. In the first step, we initialize k cluster centroids and sample $B = \{b_1, b_2, \ldots, b_m\} \subset X$ points at random from the data set X, where m is the mini-batch size. In the second step, we update the cluster centroids by gradually moving the centroids. For each sample b in the mini-batch, we update the centroids by taking the average of b and the previous points assigned to the centroid. By doing so, we move the centroid with a decreasing rate over time. We repeat the first and second steps until convergence is met (e.g. change of loss is less than a set threshold).

The main advantage of mini-batch k-means clustering over standard k-means is that the convergence time is lower. By using mini-batches, we drastically reduce the computational requirement for converging to a local solution and, the results of mini-batch k-means clustering tend to only be slightly worse than the standard algorithm.

2.1.2.3 k-medoids clustering

K-medoids clustering is an alternative to the standard k-means clustering algorithm [Kaufman and Rousseeuw, 1990; Bishop, 2006, p. 427 - 428]. K-medoids clustering uses data points for its cluster centroids and works with any dissimilarity measure. A *medoid* of a cluster is a data point where the average dissimilarity between the medoid and all other data points in the same cluster is minimal. We refer to [Kaufman and Rousseeuw, 1990; Bishop, 2006, p. 427 - 428] when explaining k-medoids clustering.

To solve the k-medoids problem efficiently, we use the *Partitioning Around Medoids* (PAM) algorithm. Similar to the standard k-means clustering algorithm, PAM consists of two main stages, namely the *build-* and *swap* stages. In the build stage, we greedily select k of the n

data points to be the initial cluster medoids, which we denote $M = \{m_1, m_2, \ldots, m_k\} \subset X$. To select M initially, we want to minimize the dissimilarity between the cluster medoids and points in the same cluster. In other words, initially, we set the first medoid m_1 to be the data point such that the dissimilarity between then medoid and all other data points is minimal. Then, for all proceeding medoids $(m_2, \ldots, m_k)$, we look for medoids such that the dissimilarity between the additional medoid, the data points in the same cluster as the new medoid, and all other medoids (and its cluster data points) is minimal. We repeat this process until we have k medoids. Following, the swap stage consists of iteratively swapping out the k medoids with other data points from the data set, if it minimizes the overall dissimilarity. The algorithm terminates if by swapping the medoids with other data points we do not get lower dissimilarity. Finally, we illustrate the use of k-medoids clustering in Figure 2.4, where we see that each point is connected to its cluster by a line, signalizing the dissimilarity between the points and the cluster medoids. The black dots represent the cluster medoids.

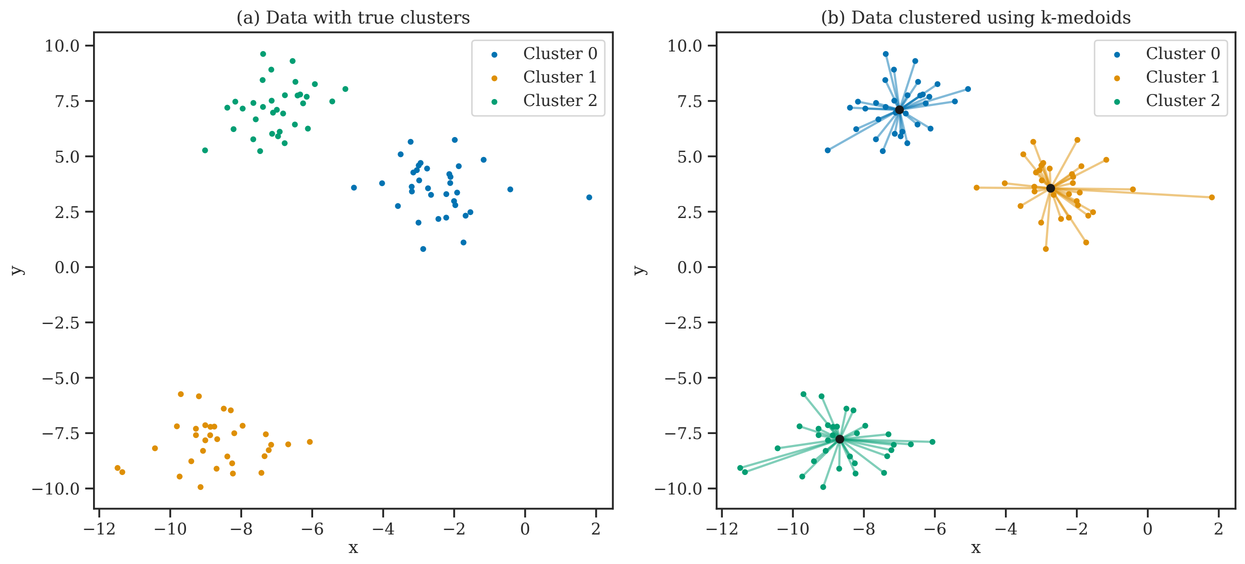

Figure 2.4: Example of clustering using k-medoids clustering on a 2-dimensional data set with 3 clusters. The lines signalize dissimilarity between points and the cluster medoids.

The main advantage of k-medoids is that it is more interpretable and robust to outliers than the standard k-means clustering algorithm since it uses actual data points as centroids. In addition to this, we may use any dissimilarity measure, whereas, in the standard k-means clustering, Euclidean distance is the only option. Even though k-medoids clustering seems to be the superior choice over k-means clustering, it suffers from the fact it is more computationally heavy to compute and thus is not always feasible to run for large data sets.

2.1.2.4 Gaussian mixture models

Gaussian mixture models (GMMs) are probability distributions which consists of a mixture of multiple *Gaussians* [Bishop, 2006, Section 9.2]. A Gaussian (i.e. normal) is a probability distribution which was several nice properties, such as mean as its mode and symmetry. In the context of cluster algorithms, we use GMMs to cluster data points by using multivariate (i.e. of higher dimension) Gaussian distributions as cluster centroids. In particular, for each cluster centroid $c_i, 1 \leq i \leq k$, we define μ_i to be the cluster mean, Σ_i to be the cluster covariances and π_i to be the *mixing coefficients*. The cluster means μ_i and covariances Σ_i determines the localization and spread for each cluster, while the mixing coefficients π_i determine how much we emphasize each cluster. To estimate the parameters $\theta = \{\mu, \Sigma, \pi\}$ of GMMs, we use the Expectation-Maximization (EM) algorithm, which is an iterative algorithm. When explaining GMMs and the EM algorithm, we refer to [Bishop, 2006, Section 9.2].

The EM algorithm starts by initializing its parameters μ, Σ, and π. There exist several methods for initializing the parameters and it is common to first run k-means clustering on the data to obtain a suitable starting point. By running k-means clustering first, we compute the initial parameters by using statistics from the result of k-means. Furthermore, the EM algorithm consists of two main stages, namely *expectation* and *maximization*. In the expectation stage, we compute the responsibilities for each data point in X using the current set of parameters. With responsibilities, we mean how much each Gaussian is responsible for explaining a single data point in X. Next, in the maximization step, we use the responsibilities from the expectation step to update the parameters such that the likelihood $\mathrm{P}(X|\theta)$ is maximal. The likelihood $\mathrm{P}(X|\theta)$ tells us how good the set of parameters θ fits our data X. The exact derivation and update rules for each parameter are left out and we refer the reader to [Bishop, 2006, Section 9.4] for more details. Once a suitable threshold is met with respect to $\mathrm{P}(X|\theta)$, the algorithm terminates and we converge to a set of parameters $\hat{\theta}$. Using the final parameters, $\hat{\theta}$, we predict which Gaussian (i.e. cluster) to associate for every data point $x \in X$, by selecting the Gaussian with the highest density. Finally, we illustrate the effect of GMM clustering in Figure 2.5, where we see that the different clusters have different shapes (i.e. means and covariances).

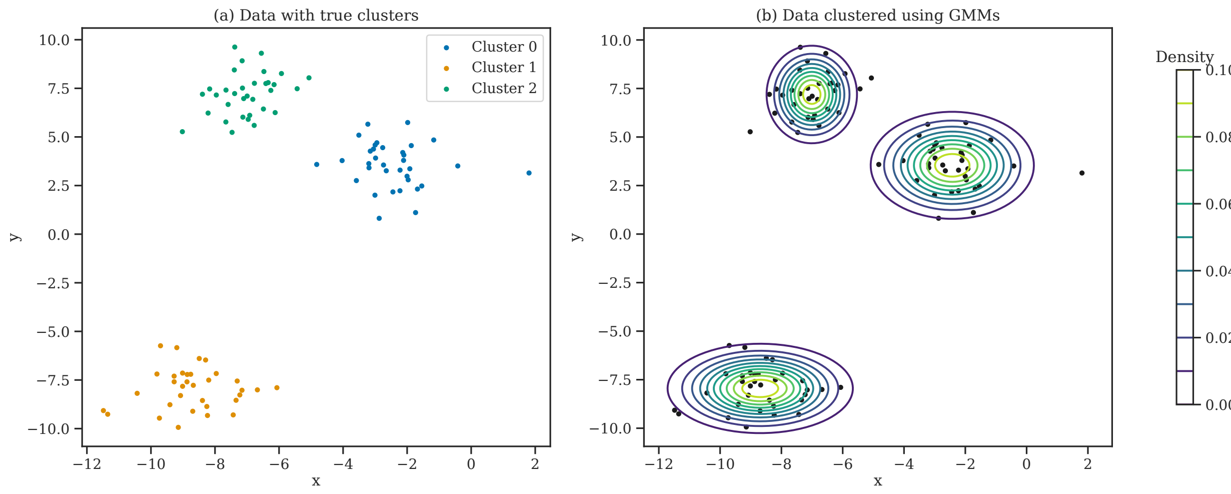

Figure 2.5: Example of clustering using GMMs on a 2-dimensional data set with 3 clusters. The different clusters have different shapes, as shown by the ellipsoids.

The main advantage of using GMMs is that clusters can be of different shapes and we get a probabilistic measure of which cluster each data point belongs to (i.e. fuzzy clustering). The convergence time of GMMs depends on the initialization of the parameters θ. If we use k-means clustering to initialize the parameters $\hat{\theta}$, then the overall convergence time is greater than simply running k-means alone. On the other hand, if we use a completely random initialization of the parameters θ, then the GMMs converges a lot faster at the risk of converging in a bad local maximum, leading to worse results.

2.1.2.5 Hierarchical clustering

Hierarchical clustering is a group of clustering algorithms that constructs clusters by recursively partitioning the data points of X in top-down or bottom-up fashion [Rokach and Maimon, 2005]. We divide the two methods of hierarchical clustering into what we call *agglomerative* and *divisive hierarchical clustering*. Furthermore, we base this sub-subsection on [Rokach and Maimon, 2005].

Using the agglomerative hierarchical clustering, each data point in X starts in its own cluster and we successively merge them until all points are in their own respective clusters. In contrast to the agglomerative method, divisive hierarchical clustering starts with all data points in X in a single cluster. Then, we divide the single cluster into smaller clusters, until each point is in its own cluster. Following, we call the output of a hierarchical clustering algorithm a *dendrogram*. We use dendrograms to represent the clustering tree structure. We illustrate an example of a dendrogram in Figure 2.6.

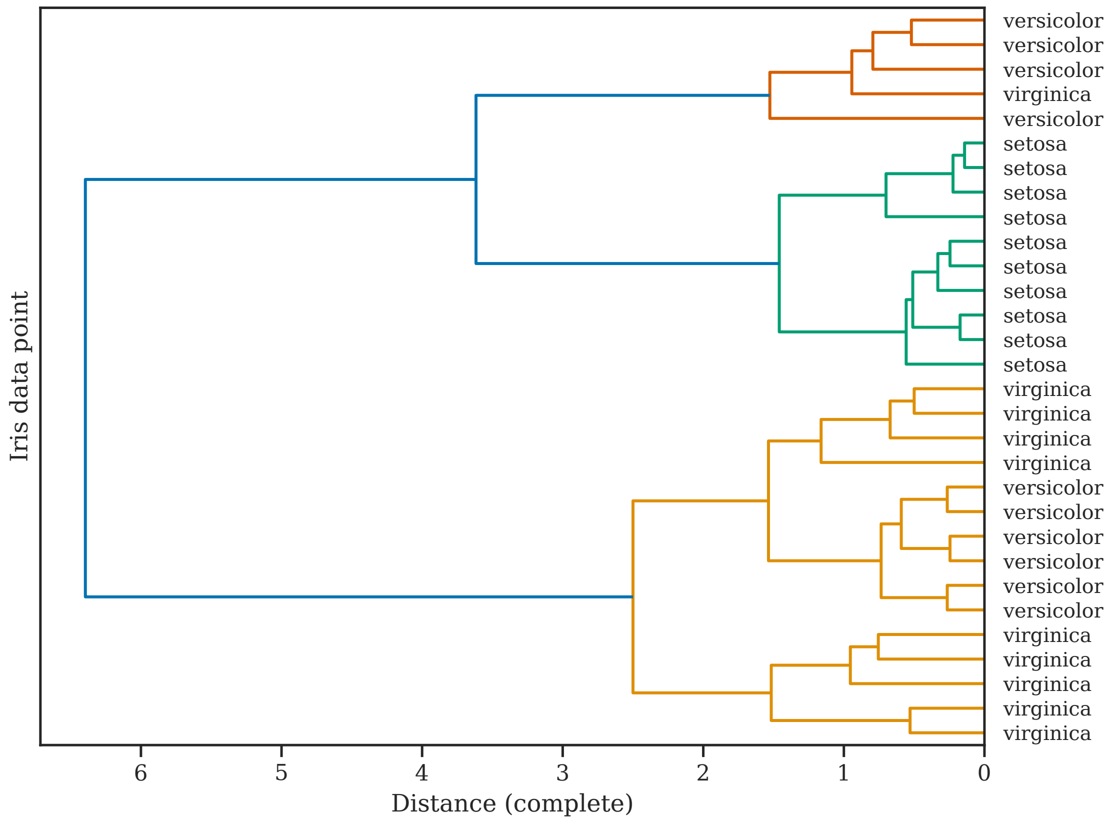

Figure 2.6: Complete-linkage clustering on a subset of the Iris data set [Anderson, 1936, Fisher, 1936], showing its dendrogram.

To merge or divide clusters, we use some similarity measure to either merge similar data points (agglomerative) or divide dissimilar data points (divisive). Exactly which data points we choose to merge or divide depends on which criterion we want to optimize. There exist several different criteria we may choose to perform hierarchical clustering. Below we list some of the most common ones and mention some pros/cons for each criterion.

- ***Single-linkage* clustering** — combines two clusters that contain the closest pair (i.e. largest similarity) of elements that do not yet belong to the same cluster as each other.

  - Single-linkage clustering tends to produce clusters of long chains, which can lead to the clustering of data points which in reality are far apart from each other.
  - Single-linkage clustering is fast to use for big data sets and can create clusters of different shapes and sizes.
  - Single-linkage clustering is sensitive to noise.

- ***Complete-linkage*** **clustering** — combines two clusters that contain the furthest pair (i.e. smallest similarity) of elements that do not yet belong to the same cluster as each other.
  - Complete-linkage clustering has bias towards spherical clusters.
  - Complete-linkage clustering works well on data with noise.
  - Complete-linkage clustering tends to split large clusters.

- ***Average-linkage*** **clustering** — combines two clusters such that the average pairwise distance of the new cluster is minimum.
  - Average-linkage clustering has bias towards spherical clusters.
  - Average-linkage clustering works well on data with noise.

- ***Ward-linkage*** **clustering** — combines two clusters such that the variance of the new cluster is minimum [Joe H. Ward, 1963].
  - Ward-linkage clustering has bias towards spherical clusters.
  - Ward-linkage clustering works well on data with noise.

Overall, hierarchical clustering is a great clustering algorithm for partitioning the data in a tree fashion. By performing hierarchical clustering, we use the resulting dendrogram to determine the number of clusters by cutting it at a certain distance threshold. In the example from Figure 2.6, a suitable cut could be at distance equal to 3, leading to 3 clusters. In addition to this, different choices of linkages can lead to different clusterings. For this reason, we should test multiple linkages to figure out what fits the data the most.

2.1.2.6 Spectral clustering

Spectral clustering is a clustering algorithm that first reduces the dimensionality of the data set and then applies a clustering algorithm [Andrew Ng and Weiss, 2002]. In particular, spectral clustering uses the eigenvalues of the *affinity matrix* (e.g. a similarity matrix using pairwise Euclidean distances) of the data X to reduce its dimensionality before applying some common clustering algorithm, such as k-means clustering (see Section 2.1.2.1). We base this sub-subsection on [Andrew Ng and Weiss, 2002].

Imagine that we want to cluster the data into k clusters. Spectral clustering starts with the construction of the affinity matrix A. We typically use some similarity measure to create pairwise distances between data points to create such an affinity matrix. Then, we compute the *graph Laplacian* $L = D - A$, where D is a diagonal matrix with $D_{ii} = \sum_j A_{ij}$ and A is the affinity matrix. The graph Laplacian L is simply a matrix representation of a graph, and in our case, the similarities between data points in X. Next, we compute the eigenvectors of L, and using these eigenvectors we get a lower-dimensional space of the original data X (from d dimensions to k). Finally, we use a clustering algorithm, such as k-means clustering, on the eigenvectors of L to get the final clustering.

The main advantage of spectral clustering is that it performs a dimensionality reduction on the data before applying a clustering algorithm. The dimensionality reduction can make the clustering algorithm less prone to noise and yield better results. Unfortunately, the computational requirement of spectral clustering is rather high, and for big data sets, it is simply infeasible.

2.1.2.7 HDBSCAN

Clusters come in different shapes and sizes, and real-life data is rather noisy. *DBSCAN* is a density-based algorithm that handles clusters of different shapes and sizes and is robust to noise [Ester et al., 1996]. It, however, is only able to produce a "flat" clustering using some global threshold parameter. *HDBSCAN* is a generalization of DBSCAN and improves on it by creating a complete density-based clustering hierarchy [Campello et al., 2013], automatically extracting flat clusters. HDBSCAN is different from the other clustering algorithms we have seen so far, as it can perform clustering without predetermining the number of clusters beforehand and can mark data points a noise. To fully understand the HDBSCAN algorithm, we introduce its key concepts and then explain how the algorithm works in practice. We base this sub-subsection on the "How HDBSCAN Works" article from [McInnes, 2016].

HDBSCAN is a density-based clustering algorithm and, for this reason, requires an inexpensive density estimation method to be efficient. Using k-nearest neighbours, the authors of HDBSCAN estimate the density efficiently. In particular, we first define the *core distance* of a data point $x \in X$ to be the distance to the *minPts*-nearest neighbour (including x), which we denote $d_{core}(x)$. *minPts* is a hyperparameter and controls how conservative we want the clustering to be; the larger *minPts*, the more "noisy" data points. To further spread apart data points that have low density, we define the *mutual reachability distance* metric (MRD) as

$$d_{mreach}(x, y) = \max\left\{d_{core}(x), d_{core}(y), d(x, y)\right\}, \tag{2.3}$$

where $d(x, y)$ is the distance between data point x and data point y using the original distance metric. Under the MRD metric, data points in dense regions do not change their distances, but for sparse data points, the distances change such that they are at least their core distance to other points.

Next, using the MRD metric, we find dense areas in the data. To find such areas, we create a *minimal spanning tree* (MST) where each node represents a data point $x \in X$ and edges connecting pairs of nodes has a weight equal to the MRD between the two nodes. By using an MST, we get a graph with the minimal set of edges between nodes such that the weight between the nodes is minimal. Additionally, if we drop exactly one edge of the graph, we disconnect it; for each pair of nodes, we connect them by exactly one edge. Using these two facts, we create a clustering hierarchy in a single-linkage clustering manner. First, we sort the weights of the edges of the MST in increasing order. Following, we iterate over the edges of the MST and merge data points into clusters (note that each data point is its own cluster initially). Now, from the hierarchical clustering, we are left with a dendrogram, which we illustrate in Figure 2.7. We are now are left with a critical question: How should we define the cut to get a flat clustering?

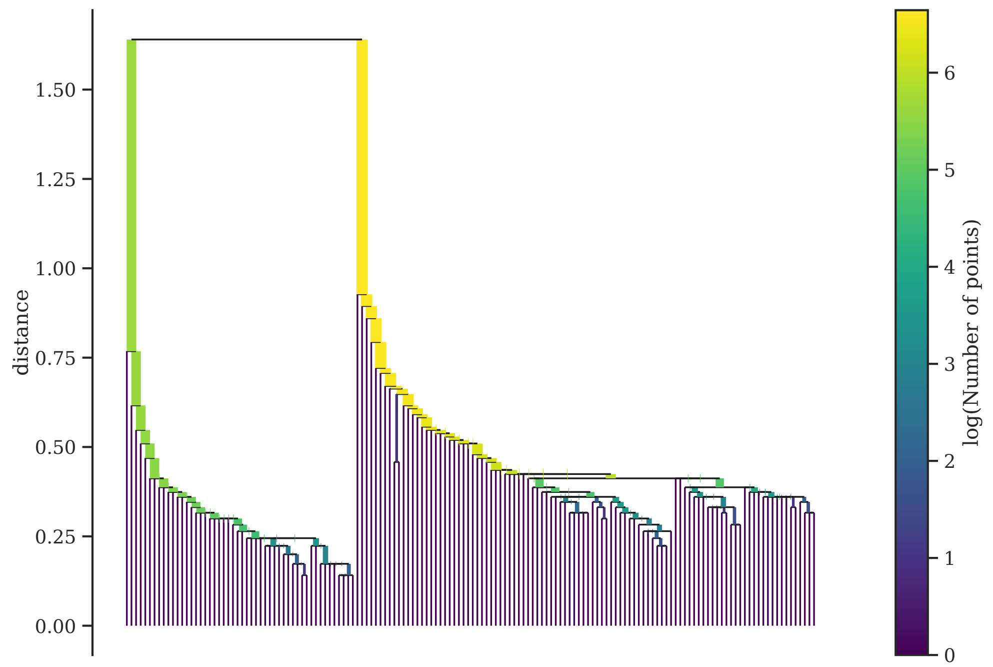

Figure 2.7: Single-linkage dendrogram plot from HDBSCAN on the Iris data set [Anderson, 1936, Fisher, 1936].

Dendrograms can be difficult to interpret, especially once we reach a certain number of data points. For this reason, the authors of HDBSCAN *condense* (or *compact*) the dendrogram from the hierarchical clustering such that they obtain a flat clustering. First, we define the notion of *minClusterSize*, which is a hyperparameter controlling the minimal cluster size at any time. Following, we walk down the dendrogram, starting from the root cluster, and at each split, we check whether or not the new cluster has at least *minClusterSize* data points in it. If the new cluster has at least *minClusterSize* data points in it, we let that cluster be in the tree. If the new cluster has less than the *minClusterSize* in it, then we let the parent cluster identify the new cluster and we remove the node from the tree. As we walk through the dendrogram to condense it, we also include at what distance clusters merge into the parent cluster, i.e. "fall out of clusters". We illustrate with an example of a condensed dendrogram in Figure 2.8.

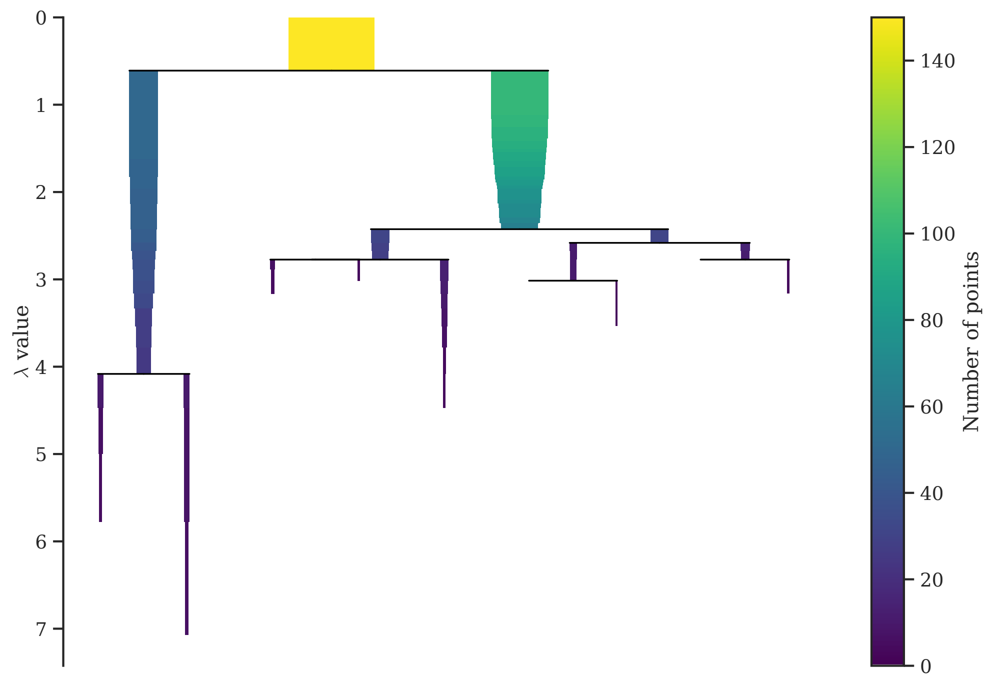

Figure 2.8: Condensed dendrogram plot from HDBSCAN on the Iris data set [Anderson, 1936, Fisher, 1936].

Now, to define the flat cut in a condensed diagram, we select the clusters such that the largest total area of "ink" is maximal, under an additional constraint that we do not select clusters that are descendants of an already selected cluster. Furthermore, we mark any clusters which we do not select in the previous step as noise, as they are merely artefacts of the initial hierarchical clustering. We then decide the final clustering, which we illustrate in Figure 2.9.

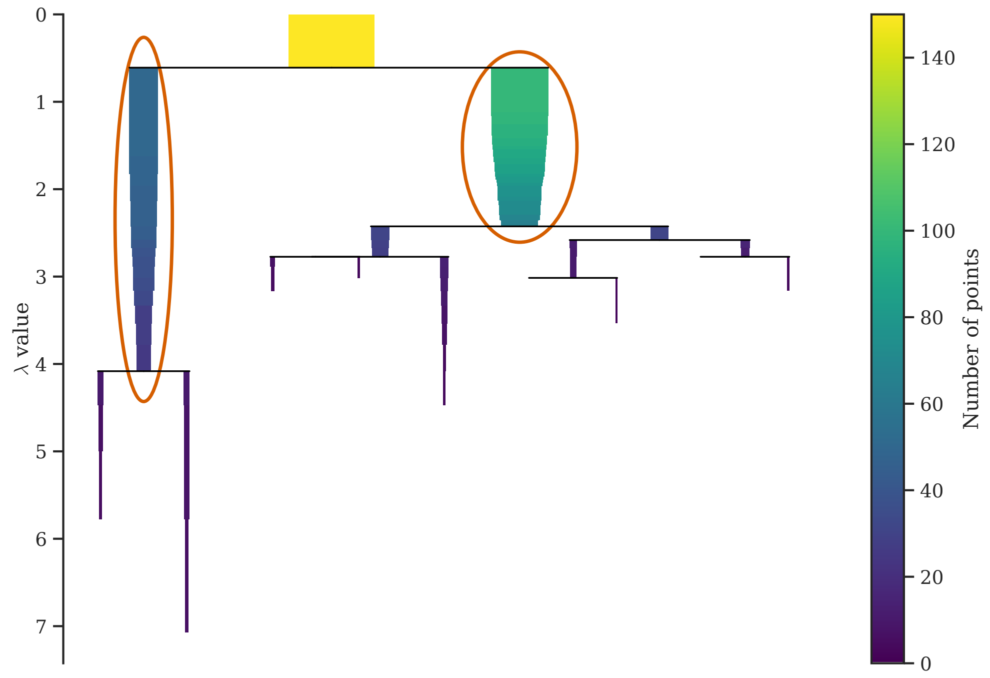

Figure 2.9: Condensed dendrogram plot from HDBSCAN on the Iris data set [Anderson, 1936, Fisher, 1936]. We show the "final cut" in the red circles and consider all data points below the clusters with the red circles as noise.

The main advantage of HDBSCAN is that it is able to find the number of clusters automatically, that we can have different shapes of clusters and are able to mark data points as noise. Dealing with noisy data points can be a challenge, and depending on how we treat them (exclusion, each noisy data point in its own cluster, etc.), it may lead to different results.

2.1.2.8 ToMATo

Topological Mode Analysis Tool (ToMATo) is a clustering algorithm that uses concepts from topological data analysis (TDA) [Oudot, 2015, p. 118-131]. In particular, ToMATo uses the concepts of persistence diagrams (see Section 2.3.2) and prominence to perform clustering. We divide the ToMATo clustering algorithm into three parts: density estimation and neighbour graph creation (1), mode-seeking (2) and merging (3). Furthermore, we refer to [Oudot, 2015, p. 118-131] when explaining the algorithm.

First (1), we use any density estimation scheme to estimate the density of our data. A common choice is to use kernel density estimation with some kernel (e.g. Gaussian). We denote the density of a data point $x_i \in X, i = 1, 2, \ldots, n$ as $\tilde{f}(x_i)$. In addition to density estimation, we compute a neighbourhood graph G to determine the neighbours of data points. In the graph G, each vertex is a data point and edges represent neighbours. To compute G, it is common to use a k-nearest neighbour graph, where k represents the number of neighbours.

Using the density estimator $\tilde{f}$ and neighbourhood graph G, we compute the initial clusters of ToMATo by performing mode-seeking (2). First, we sort the vertices of G by decreasing $\tilde{f}$-values and iterate over them. At each vertex i, compare the $\tilde{f}$ values of vertex i and its neighbours. If $\tilde{f}(x_i)$ is greater than $\tilde{f}$ of its neighbours, we then denote vertex i as a peak of $\tilde{f}$. Otherwise, we connect vertex i to the neighbour with the greatest $\tilde{f}$-value. By doing so, we create a spanning forest, where each spanning tree represents peaks of the underlying true density function. In the next step, we use this spanning forest and merge the trees to obtain a clustering.

The last step is the merging (3) of the spanning forest from (2). To do this, ToMATo iterates over the vertices of G again (in the same order as in (2)) and we use a *union-find* data structure to keep track of the spanning trees we merge. We denote the union-find data structure as $\mathcal{U}$. The entries $e \in \mathcal{U}$ correspond to the union of spanning trees. The root of an entry $r(e)$, is the vertex in e whose $\tilde{f}$-value is the highest, i.e. a local peak of $\tilde{f}$ in G. Now, iterating over the vertices of G, we check whether or not vertex i is a peak of $\tilde{f}$. If vertex i is a peak of $\tilde{f}$, i.e. root of some spanning tree S, we create a new entry e in $\mathcal{U}$, in which we store S. The root of entry e is set to the vertex itself, i.e. $r(e) = i$. If vertex i is not a peak of $\tilde{f}$, it means that it belongs to some existing entry $e_i \in \mathcal{U}$ and we look for potential merges of e_i with other entries in $\mathcal{U}$. In particular, we iterate over neighbours $e \in \mathcal{E}$, $e \neq e_i$, of i in G and check whether $\min\left\{\tilde{f}(x_{r(e)}), \tilde{f}(x_{r(e_i)})\right\} < \tilde{f}(x_i) + \tau$, where τ is the *prominence* threshold parameter. In other words, we check whether or not two entries have different $\tilde{f}$-value and at least one of them has root with less than τ prominence. If this is true, we merge e and e_i into a single entry in $\mathcal{U}$, i.e. $e \cup e_i$, and we merge the entry with the lower root into the one with the higher root.

Once the merging step is complete, we are left with a union-find structure $\mathcal{U}$. For every entry e of $\mathcal{U}$, we connect them to its parent entry $p(e)$ such that $\tilde{f}(x_{p(e)}) > \tilde{f}(x_e)$. In other words, by iteratively searching for the topmost parent, we determine which cluster we connect each entry (i.e. data point) to. We see the ToMATo clustering algorithm as a combination of mode-seeking (from step (2)) and hierarchical clustering (from step (3)). As a result of the hierarchical structure, we can visualize when entries in $\mathcal{U}$ merge into other entries, and thus, explain the lifespans of entries. More precisely, an entry in $\mathcal{U}$ is "born" when we create it in $\mathcal{U}$ and "dies" when we merge it into another entry with a higher root. Using persistence diagrams (see Section 2.3.2) we can explain the lifespans of entries and determine which entries live the longest (i.e. entries that never dies). We use the persistence diagram to determine which value for τ we should use. Different values of τ results in different numbers of clusters. In practice, let $\tau = +\inf$ and use the persistence diagram of $\mathcal{U}$ to find a suitable threshold $\hat{\tau}$ such that we get the number of clusters we want. Then, we run ToMATo again using $\hat{\tau}$ as the threshold parameter to get the final clustering. Finally, we motivate the use of ToMATo in Figure 2.10, where we see from the persistence diagram in (b) that ToMATo found 5 clusters, but 3 of them are significantly far from the diagonal (i.e. high prominence). This indicates that the data set should consist of 3 clusters, and thus, we run ToMATo again setting the prominence threshold such that we get 3 clusters.

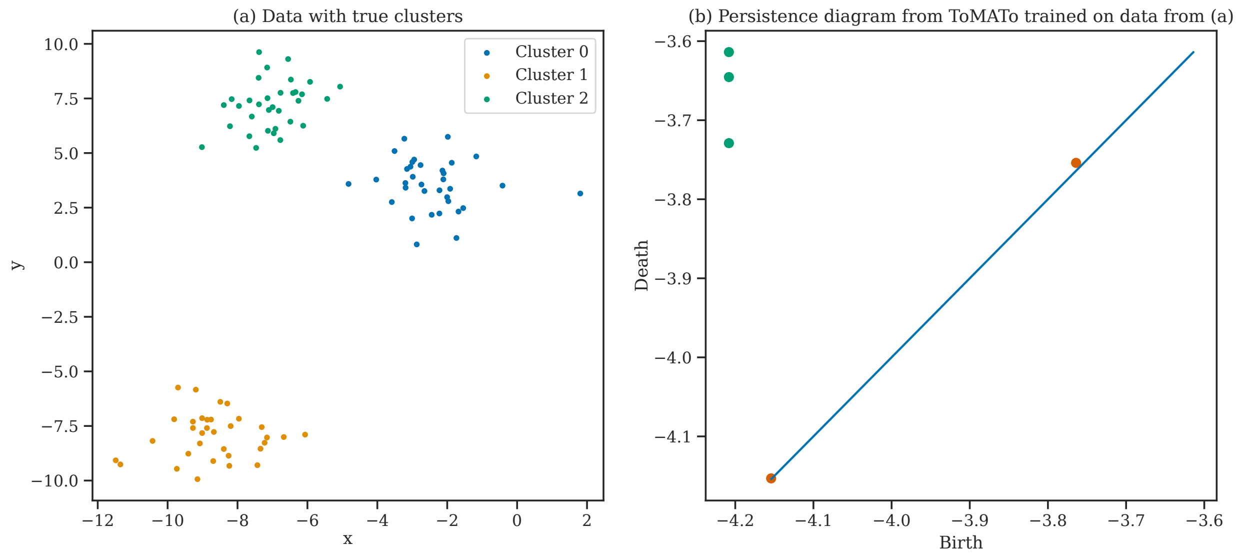

Figure 2.10: Example of clustering using ToMATo on a 2-dimensional data set with 3 clusters. The persistence diagram in (b) shows the three prominent clusters (green dots).

What is great about ToMATo is that it gives us a way to determine the numbers of clusters automatically (e.g. using some heuristic on the first persistence diagram to determine $\hat{\tau}$). In addition to this, ToMATo works with any metric, as long as we can create a neighbourhood graph (e.g. using Euclidean distance or similar metrics).

2.1.2.9 Comparison of clustering algorithms

In Section 2.1.2, we describe several algorithms for clustering data. We tabularize the strengths and weaknesses for each algorithm in Table 2.1. Even though some algorithms might tick off more properties than others, it does not mean that it is a perfect algorithm for all types of data sets. In particular, we have to perform cluster validation to evaluate the results from various cluster algorithms to figure out what algorithm works the best. We explain how to validate results from cluster algorithms in Section 2.1.3.

	Algorithm							
Property	k-means	MB k-means	k-medoids	GMMs	Hierarchical	Spectral	HDBSCAN	ToMATo
Practical for large data sets	✗	✗		✗	✗		✗	✗
Determining the number of clusters automatically					✗		✗	✗
Cluster centroids as data points			✗					
Different clusters shapes				✗	✗		✗	✗
Hierarchical clustering					✗		✗	
Robust against nosy data sets			✗			✗	✗	✗
Can identify noisy/anomalous data points							✗	✗
User-defined distance metric			✗		✗	✗	✗	✗

Table 2.1: Comparison of various properties for each clustering algorithm we describe in Section 2.1.2.

2.1.3 Cluster validation

After we use clustering algorithms to perform clustering, we evaluate the result to find the best set of hyperparameters and/or clustering algorithm. This raises the question: Which clustering algorithm performs the best on our data? Thankfully, there exist a handful of various methods to tackle this task. In particular, we differentiate between *internal* and *external cluster validation algorithms*. Internal cluster validation algorithms assess the performance of the clustering using statistics of the data, without knowing the true labels at hand. External cluster validation algorithms, on the other hand, use the knowledge of the true labels. In this thesis, we only use internal cluster validation algorithms, as we are mostly working with data where we do not know the labels beforehand. Recall that, in most clustering algorithms, we want the average distance between any two data points in the same cluster to be as small as possible (compactness); and the average distance between any two data points from different clusters to be as large as possible (separation). Internal cluster validation algorithms usually reflect compactness or separation of clusterings. In the next sub-subsections, we explain some common choices of internal cluster validation algorithms and discuss their strengths and weaknesses. Furthermore, we will use the cluster validation methods we explain below when we analyze word embeddings (Section 2.2) in Chapter 3.

2.1.3.1 Silhouette Coefficient

The *Silhouette Coefficient* is an internal cluster validation method that measures the goodness of clusterings [Kaufman and Rousseeuw, 1990, p. 87]. In particular, the Silhouette Coefficient measures how similar data points are to their own cluster (compactness) when we compare to data points from other clusters (separation). The Silhouette Coefficient ranges from -1 to 1, where the best value is 1 and the worst value is -1. Values near 0 indicate that we have overlapping clusters (i.e. low separation). We base this sub-subsection on [Kaufman and Rousseeuw, 1990, p. 87].

For any data point x_i in the cluster C_i, we compute the mean compactness $a(i)$ as the average distance $d(i,j)$ between data point i and all other data points in C_i, i.e. $j \in C_i, i \neq j$. More formally, we define the mean compactness $a(i)$ as

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i,j), \tag{2.4}$$

where $|C_i|$ is the number of data points in cluster C_i. Smaller values of $a(i)$ indicate better compactness of clusters, and thus, better cluster assignments. Following, for any data point x_i in the cluster C_i, we compute the mean separation $b(i)$ as the smallest distance from i to any other cluster in which i does not belong. The mean separation $b(i)$ is defined as

$$b(i) = \min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i,j). \tag{2.5}$$

Large values of $b(i)$ indicate that the clusters have good separation and that data points in C_i are a good fit. Now, using the definitions of compactness and separation, we define the *silhouette* of data point i as

$$s(i) = \begin{cases} \frac{b(i)-a(i)}{\max(a(i),b(i))} & \text{if } |C_i| > 1 \\ 0 & \text{if } |C_i| = 1 \end{cases}. \tag{2.6}$$

Taking the average of all silhouettes $s(i)$, we define the *mean Silhouette Coefficient* as

$$SC = \frac{1}{n}\sum_{i=1}^{n} s(i), \tag{2.7}$$

where n is the number of data points in our data (e.g. n-dimensional data X).

The main advantages of the mean Silhouette Coefficient are that it is simple, fast to compute and has a defined range from -1 to 1. The mean Silhouette Coefficient struggles with overlapping clusters (where $SC \approx 0$).

2.1.3.2 Davies-Bouldin Index

The *Davies-Bouldin Index* (DBI) is an internal cluster validation method for evaluating results of clustering algorithms [Davies and Bouldin, 1979]. Similar to the Silhouette Coefficient, DBI measures the compactness and separation of clusters to measure the goodness of fit. We base this sub-subsection on [Davies and Bouldin, 1979].

To measure the compactness of clusters, DBI introduces the notion of *scatter within a cluster*, which we denote S_i. We compute S_i by taking the mean of the sum of the distances to the cluster centroid of a particular data point i. More formally, we define S_i as

$$S_i = \left(\frac{1}{|C_i|} \sum_{x_j \in C_i} |x_j - \tilde{C}_i|^p \right)^{1/p}, \tag{2.8}$$

where C_i is the cluster we associate to data point i, $|C_i|$ is the number of data points in cluster C_i, $\tilde{C}_i$ is the centroid of cluster C_i and p denotes the power of the L^p distance. A common choice is to set $p = 2$, leading to Euclidean distances. A low value of S_i indicates that the compactness of cluster C_i is good. To compute the separation of clusters, DBI computes separation between clusters C_i and C_j by taking the distance between cluster centroids $\tilde{C}_i$ and $\tilde{C}_j$. We define the cluster separation $M_{i,j}$ as

$$M_{i,j} = ||\tilde{C}_i - \tilde{C}_j||_p, \tag{2.9}$$

where p denotes the power of the L^p distance. High values of $M_{i,j}$ indicate that we separate the clusters well. Combining the notion of cluster compactness S_i and separation $M_{i,j}$, we define the measure of cluster goodness $R_{i,j}$ as

$$R_{i,j} = \frac{S_i + S_j}{M_{i,j}}. \tag{2.10}$$

To create a clustering measure that is symmetric and non-negative, we define $R_{i,j}$ as such. Low values of $R_{i,j}$ indicate that we have compact clusters with high separation. For a particular cluster C_i, DBI measures $R_{i,j}$ for all other clusters C_j, $j \neq i$, and uses the largest value of $R_{i,j}$, i.e. the worst-case scenario, to compute the index. More formally, we define DBI as

$$DBI = \frac{1}{n}\sum_{i=1}^{n} \max_{j \neq i} R_{i,j}, \tag{2.11}$$

where n is the number of data points in our data (e.g. n-dimensional data X). The main advantage of DBI is the fact that is fast to compute and has non-negative values. Since DBI does not have any upper bound, it is more difficult to interpret the values, when we for instance compare it to Silhouette Coefficient.

2.1.3.3 Caliński-Harabasz Index

The *Caliński-Harabasz Index* (CHI) is an internal cluster validation method for evaluating clustering algorithms [Caliński and JA, 1974]. CHI measures compactness and separation of clusters to measure the goodness of fit of a particular clustering. We base this sub-subsection on [Caliński and JA, 1974].

To measure the compactness of a clustering, CHI computes the *sum-of-squares within cluster*

(SSW) as

$$SSW = \sum_{i=1}^{n} ||x_i - \tilde{C}_i||^2, \tag{2.12}$$

where n is the number of data points in our data, x_i is the ith data point and $\tilde{C}_i$ is the centroid of the cluster we associate to x_i. Small values of SSW indicate that we have compactly clustered data points. To measure the separation of clusters, CHI computes the *sum-of-squares between clusters* (SSB) as

$$SSB = \sum_{j=1}^{k} |C_j| \cdot ||\tilde{C}_i - \bar{X}||^2, \tag{2.13}$$

where k is the number of clusters, $|C_j|$ is the number of data points in cluster C_j and $\overline{X}$ is the centre of all data points (i.e. mean). Large values of SSB indicate that we have a good separation of the clusters. Finally, we compute the CHI by multiplying the ratio of SSB to SSW with a weight depending on n and k. More formally, the CHI is defined as

$$CHI = \frac{SSB}{SSW} \times \frac{n-k}{k-1}. \tag{2.14}$$

Since we define CHI as the ratio between SSB and SSW, large values indicate better clustering. The main advantage of CHI is that it is fast to compute. Similar to Davies-Boundin Index, CHI is harder to interpret than Silhouette Coefficient for example, since the values do not have any upper limit.

2.1.4 Dimensionality reduction methods

Working with data in high dimensions can be a difficult task. Imagine that we have gathered some data, containing several features, and we would like to deepen our understanding of it. One approach we could do would be to visualize each feature by plotting them against each other, looking for bivariate relationships. Unfortunately, this is a rather cumbersome task and is hard to employ once a certain number of features is met (e.g. 10 features). Thankfully, *dimensionality reduction methods* can help us to reduce the dimensionality of data into some lower dimension, preserving relevant properties of the original data in the process. A typical application of dimensionality reduction methods is to lower the dimensionality to 2 or 3 such that we can visualize the data at ease. We introduce two dimensionality reduction methods and explain how they work. In both sub-subsections below, we assume that we have some data $X \in \mathbb{R}^{n \times d}$ and that we want to reduce the dimensionality to some chosen hyperparameter k. Furthermore, we will use the dimensionality reduction methods which we explain below when we analyse clustering of word embeddings (Section 2.2) in Chapter 3 and prediction of polysemous words in Section 3.3.4.

2.1.4.1 Principal Component Analysis

Principal Component Analysis (PCA) is one of the most common dimensionality reduction methods [Jolliffe, 2002]. PCA performs a linear mapping of the original data $X \in \mathbb{R}^{n \times d}$ to a (possibly) lower-dimensional space $Y \in \mathbb{R}^{n \times k}$ ($k \leq d$) such that the variance is maximal in Y. We typically select k to be a relatively low value, e.g. 2 or 3, if we would like to visualize it using plots. Another common value for k is to set it such that a specific variance threshold is met, e.g. explaining 90% of the variance of the data. We refer to [Jolliffe, 2002] when explaining PCA.

The PCA algorithm consists of several steps, and following, we explain each step. First, compute the mean of X, which we denote $\overline{X} = \frac{1}{n}\sum\limits_{i=1}^{n} x_i$, and subtract it from X, i.e. $X' = X - \overline{X} = \{x'_1, x'_2, \dots, x'_n\}$. Next, we compute the covariance matrix of X', which we denote $C \in \mathbb{R}^{d \times d}$, and we compute the corresponding eigenvectors and eigenvalues of C, which we denote $C_{\text{eig}} = \{c_1, c_2, \dots, c_d\}$. Furthermore, we sort the eigenvectors C_{eig} by its corresponding eigenvalues in a descending manner. By doing so, we get a set of eigenvectors such that the first eigenvector corresponds with the largest value, the second eigenvector corresponds with the second largest value and so forth. We denote this set of eigenvectors as $C'_{\text{eig}} = \{c'_1, c'_2, \dots, c'_d\}$. We then pick the k first eigenvectors of C'_{eig} and project the data onto them, essentially performing a *change of basis*. In addition to this, we add the mean of X to complete the reconstruction. We define the reconstruction of PCA as

$$Y = X'\begin{pmatrix} c'_1 & c'_2 & \dots & c'_k \end{pmatrix}^\top + \overline{X}, \tag{2.15}$$

where $Y = \{y_1, y_2, \dots, y_n\}$ represents the original data X in (a possibly) lower dimension k, such that we maximize the variance of the vectors of X. We refer the vectors $\{y_1, y_2, \dots, y_n\}$ as the principal components (PC) of X, where PC1 is y_1, PC2 is y_2 and so forth. Finally, we show an example where we apply PCA to some 2-dimensional data in Figure 2.11, where we see that PCA essentially rotates the data, maximizing the variance in the PC axes.

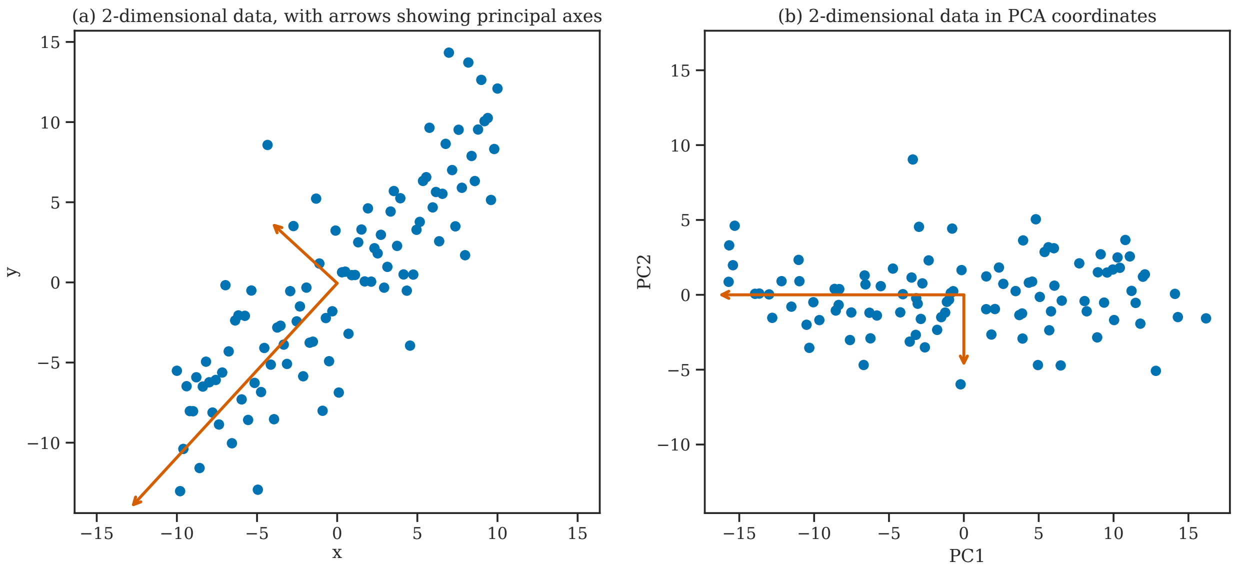

Figure 2.11: PCA applied to 2-dimensional data, showing the principal axes by the arrows.

2.1.4.2 Uniform Manifold Approximation and Projection

Uniform Manifold Approximation and Projection (UMAP) is a dimensionality reduction algorithm that, among other things, uses ideas from topological data analysis [McInnes et al., 2018]. In particular, UMAP uses a fuzzy version of simplicial complexes (see Section 2.3.1 for the definition of a simplicial complex) to create a graph representing the topological structure of the data in its original (high) dimension. To explain how UMAP works, we use the example from the "How UMAP Works" documentation page [McInnes, 2018].

Imagine that we have some data from a noisy sine, $X = \{x_1, x_2, \ldots, x_n\} \in \mathbb{R}^{n \times d}$, as we see in Figure 2.12.

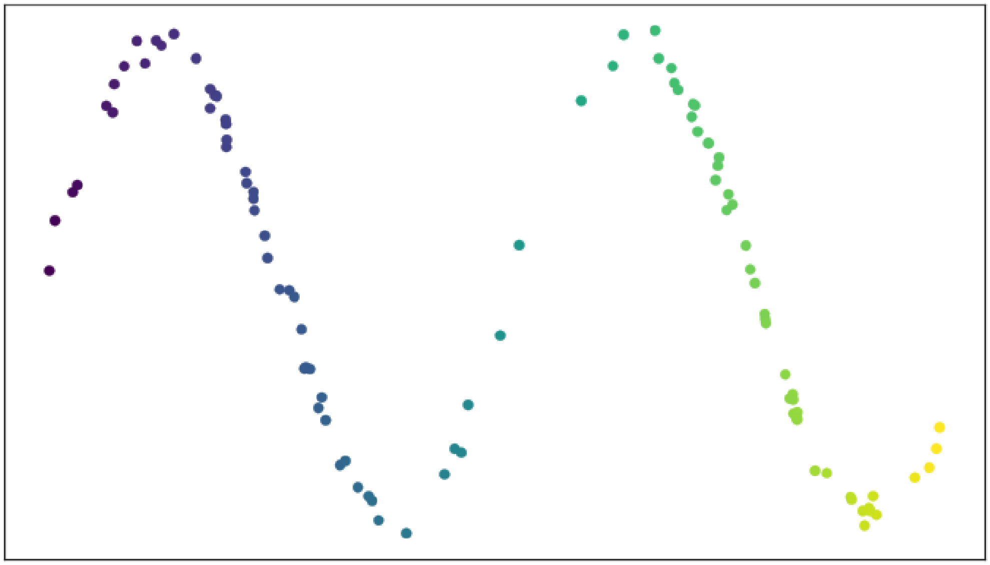

Figure 2.12: A noisy sine wave. The figure is from [McInnes, 2018].

We would like to capture the topological structure of X, and we do so by creating a simplicial complex built on X, as we see in Figure 2.13.

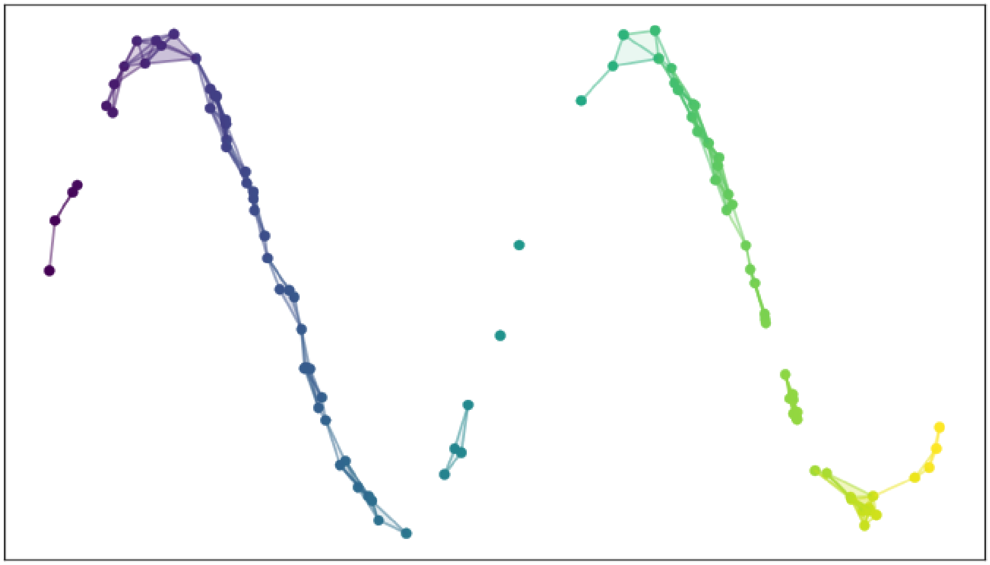

Figure 2.13: A simplicial complex built on a noisy sine wave. The figure is from [McInnes, 2018].

We would like to capture the topological structure of all data points in X and want a graph connecting all points. By using simplicial complexes, however, we see in Figure 2.13 that problems can occur, namely that not all data points have edges between them, which again disconnects the simplicial complex. This particular problem can occur when we have a too-small ϵ radius around each data point. In real-world data, the data points are typically not laying on a uniform distribution. Moreover, selecting a perfect ϵ to create a suitable simplicial complex is hard. The authors of UMAP overcome these problems by creating fuzzy open sets around each data point to create local connectivity in the graph, as we see in Figure 2.14.

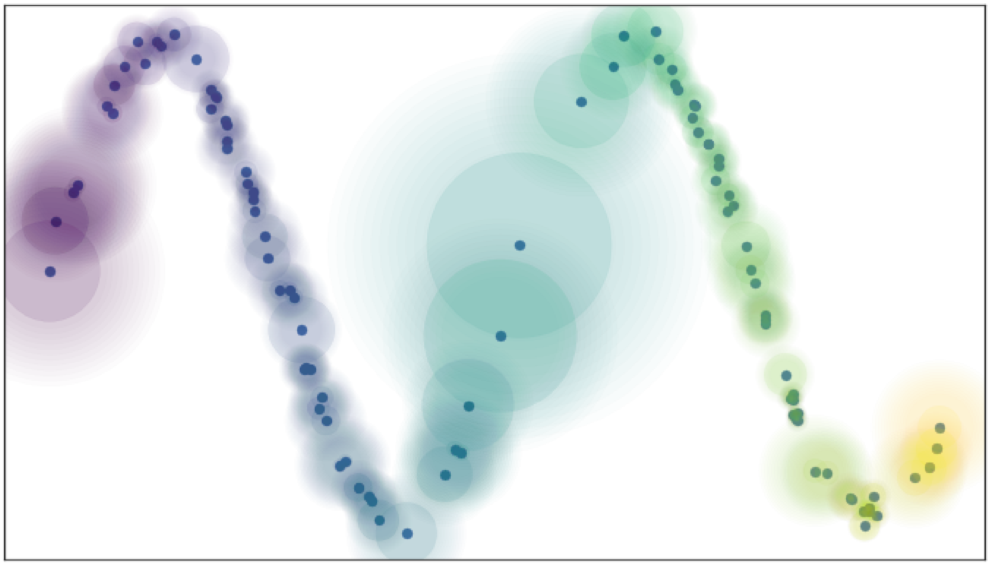

Figure 2.14: Fuzzy open sets around each data point of a noisy sine wave to create local connectivity. The figure is from [McInnes, 2018].

To create the fuzzy open sets, we compute the distance to the nearest neighbour of each data point and the level of *fuzziness* decreases in terms of the distance beyond it, starting from 1 decreasing to 0. If a data point has a fuzziness level greater than zero, then we create an edge between the two data points, with weight equal to the fuzziness level. Furthermore, we interpret the fuzziness level as the probability of the edge existing. We illustrate the connected graph in Figure 2.15.

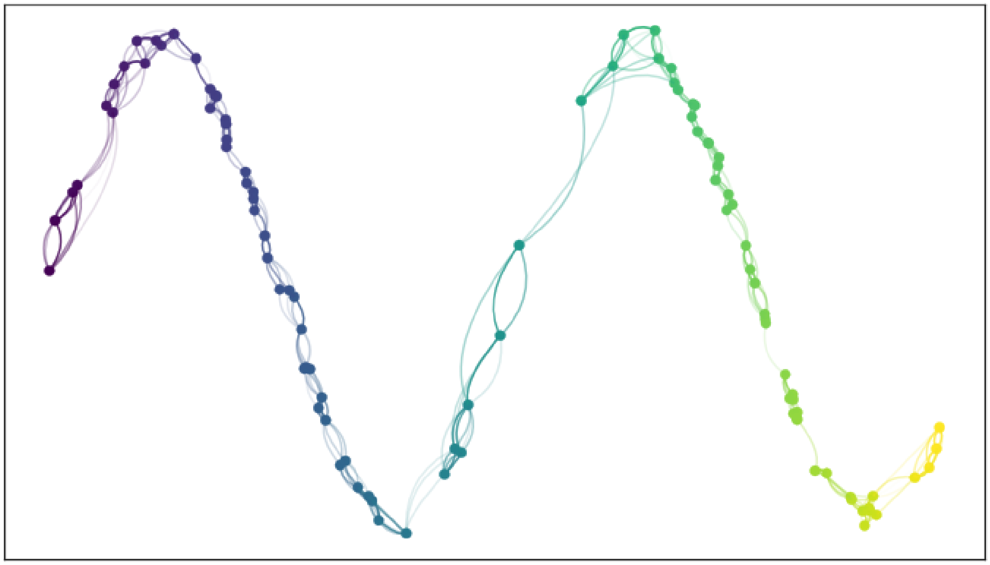

Figure 2.15: A connected graph where nodes are data points and the edges between them have the fuzziness levels as weights. The figure is from [McInnes, 2018].

To finalize the connected graph from Figure 2.15, we convert the edges between any two data points into a single edge. We do this because we want the distance between two data points a and b to be the same; currently, it depends locally on the distance to the nearest neighbour, as the fuzziness level decreases beyond it. To merge the edges between any two data points a and b, we compute the combined weight by taking the union between them $w(a)+w(b)-w(a)w(b)$. We use the newly combined weight as the weight of the single edge between a and b. If we apply this process, unioning edges together, we end up with a fuzzy simplicial complex. We show an example of a fuzzy simplicial complex in Figure 2.16.

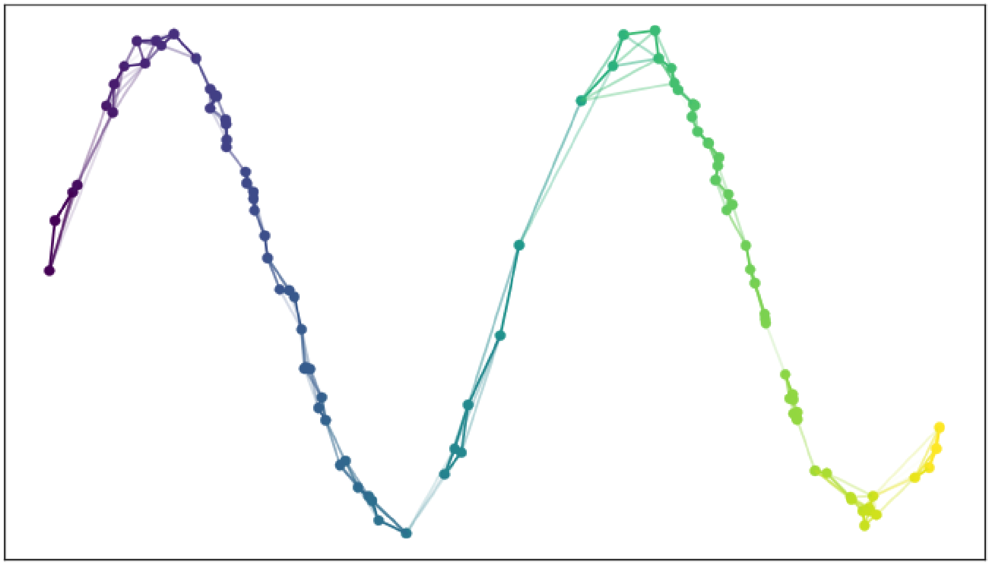

Figure 2.16: A fuzzy simplicial complex of some sine wave data. The figure is from [McInnes, 2018].

Now, using the fuzzy simplicial complex of Figure 2.16, we have a topological representation of X, capturing the topology of the manifold. We denote the set of all possible 1-simplices (i.e. edges) as E. To compute the weight of a 1-simplex (i.e. edge) e, we use $w_h(e)$ in the high dimensional case. To get a good low dimensional representation of the high dimensional fuzzy simplicial complex, we initialize a low dimensional fuzzy simplicial complex. We denote the weights of the low dimensional fuzzy simplicial complex as $w_l(e)$ for edge e. To determine the weights $w_l(e)$, we employ an iterative process where we optimize a loss function L in a gradient descent fashion (see Sections 2.1.5.5 and 2.1.5.6 for more information regarding loss functions and the gradient descent optimizer). Since we interpret the weights of the 1-simplices of E as probabilities of the edge existing (i.e. Bernoulli variables), the authors of UMAP uses cross-entropy (see Equation (2.30)) as the loss function. More formally, we define the loss function L as

$$L = \sum_{e \in E} \underbrace{w_h(e) \log\left(\frac{w_h(e)}{w_l(e)}\right)}_{\text{Attractive force}} + \underbrace{(1 - w_h(e)) \log\left(\frac{1 - w_h(e)}{1 - w_l(e)}\right)}_{\text{Repulsive force}}. \tag{2.16}$$

In the first term of Equation (2.16), we have an attractive force between the data points which e spans, pulling them together; when $w_l(e)$ is large, the distance (i.e. weight) between any two data points becomes small. In the second term of Equation (2.16), there is an opposite, repulsive force between the data points which e spans, repelling them apart; when $w_h(e)$ is small (i.e. distance in high dimensional space), $w_l(e)$ becomes small since we want to minimize the term. The process of pulling and repelling the weights makes the low dimensional representation of the data settle into a balanced state, such that it represents the high dimensional topological structure of the original data in a fairly accurate way. In practice, the UMAP algorithm uses several different tricks to optimize it, but we leave out the technical details here. We kindly refer the reader to [McInnes et al., 2018] for more details. Finally, we show an example where we compute a 2-dimensional UMAP embedding of the Iris data set [Anderson, 1936, Fisher, 1936] in Figure 2.17, where we see that UMAP can separate the classes in the Iris data set quite nicely, particularly for the *setosa* class.

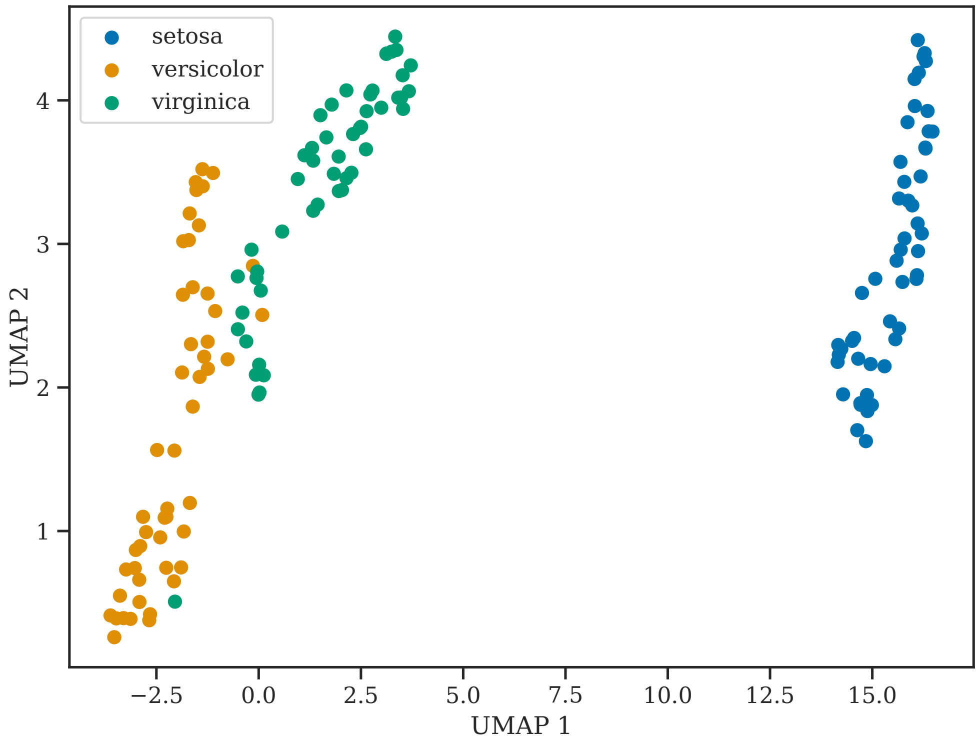

Figure 2.17: 2-dimensional UMAP embedding of the Iris data set [Anderson, 1936, Fisher, 1936].

2.1.5 Artificial neural networks

In this subsection we explain *artificial neural networks* (ANN). In particular, we focus on *multilayered neural networks* (MLNN). We base this subsection on [Aggarwal, 2018, Chapter 1] and [Rong, 2016]. Furthermore, we will use the notion of artificial neural networks when we explain the word2vec model in Section 2.2.2.5.

An *artificial neuron* (or *unit*) is a function which receives one or more inputs and a *bias*, and then sums them to produce an output. We illustrate an example of an artificial neuron in Figure 2.18, where $\{x_1, \ldots, x_K\}$ are the input values, b is the bias, $\{w_1, \ldots, w_K\}$ are the weights and y is a scalar output. We denote f as the *activation function*.

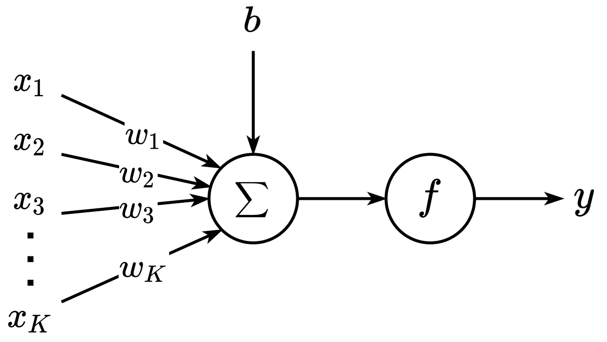

Figure 2.18: An artificial neuron that takes in a K-dimensional input x and bias b to produce some output y.

We produce the output of a single unit by applying an activation function f to the input u. More formally, we define the output of a single unit as

$$y = f(u), \tag{2.17}$$

where u is the input of the neuron. We define u as

$$u = b + \sum_{i=1}^{K} w_i x_i, \tag{2.18}$$

which is the weighted sum of the input values $\{x_1, \ldots, x_K\}$ plus the bias term, with $\{w_1, \ldots, w_K\}$ as weights.

The bias term acts as an intercept value to make the model more general and is usually set to +1. For some models (e.g. word2vec, which we introduce as an ANN in Section 2.2.2.5), we exclude the bias term for the units in the neural network, i.e. we leave b to be zero.

The choice of activation function f is typically a non-linear function. Artificial neural networks use different activation functions such as ReLU, sigmoid or tanh to learn non-linear relationships in the data. We come back to the concept of activation functions in Section 2.1.5.4.

A *layer* $\{z_j\} = \{z_1, z_2, \ldots, z_K\}$ of an artificial neural network is a collection of artificial neurons (unit). We define the layer $\{z_j\}$ using an $N \times K$-dimensional weight matrix W, a N-dimensional bias vector b and an activation function f. More formally, we define a layer $\{z_j\}$ as

$$\{z_j\} = f\left(W \cdot x + b\right), \tag{2.19}$$

where x is a K-dimensional input vector. In the following sub-subsections, we explain the different types of layers in the ANN, namely the input, hidden and output layers.

2.1.5.1 Input layer

The first layer in the ANN is the *input layer* $\{x_k\} = \{x_1, x_2, \ldots, x_K\}$. It is responsible for taking in input and passing it to the proceeding layer in the network. We illustrate the input layer in Figure 2.19, where we see that each input value x_i gets its own node.

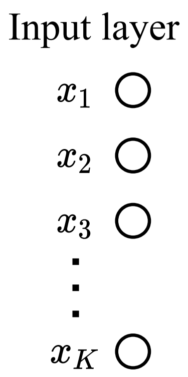

Figure 2.19: Input layer in the ANN for a K-dimensional vector x.

Following, we define the input layer more formally using Equation (2.19) in Equation (2.20). We see that the input layer does not perform any changes to the incoming data and acts as a way for feeding data into the neural network. More formally, we define the input layer $\{x_k\}$ as

$$\begin{aligned} \{x_k\} &= f_{\{x_k\}}(W_{\{x_k\}} \cdot x) \\ &= id(I_K \cdot x) \\ &= I_K \cdot x \\ &= x, \end{aligned} \tag{2.20}$$

where the weight matrix $W_{\{x_k\}}$ is the identity matrix I_K, we have no bias (i.e. bias equal to zero vector) and the activation function $f_{\{x_k\}}$ is the identity function $id(x) = x$. In the following sub-subsection, we look at the next layer in the neural network, namely the hidden layer.

2.1.5.2 Hidden layer

The *hidden layer* is the second layer in the ANN $\{h_i\} = \{h_1, h_2, \ldots, h_N\}$, and we most commonly connect it to the input layer. We note that we can, however, have multiple hidden layers in the ANN by connecting them to each other (making the neural network *deeper*). For illustration purposes, we assume that we only have one hidden layer in our neural network. We illustrate an example of a hidden layer in Figure 2.20, where we observe that we connect every unit in the input layer to the units in the hidden layer. We illustrate the connections by the lines. This is what we call *fully connected layers*, meaning that we connect every unit to each other.

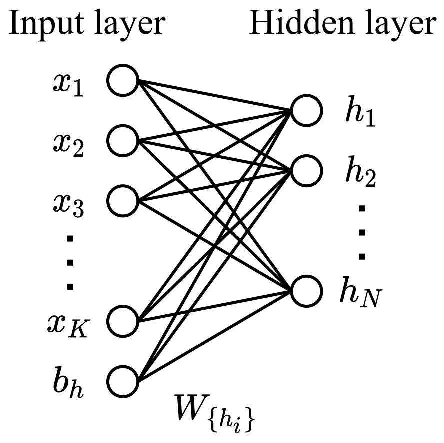

Figure 2.20: Input to hidden layer in the ANN for a K-dimensional vector x, $N \times K$ dimensional weight matrix $\{W_{h_i}\}$ and a N-dimensional hidden layer h.

We formalize the description of the hidden layer by defining it as

$$\{h_i\} = f_{\{h_i\}}(W_{\{h_i\}} \cdot \{x_k\} + b_h), \tag{2.21}$$

where $f_{\{h_i\}}$ is a user-specified activation function, $W_{\{h_i\}}$ is an $N \times K$-dimensional weight matrix and b_h is an N-dimensional bias vector. The hidden layer tries to learn a *latent* (i.e hidden) representation of the input data x. We explain how the neural network learns the latent representation when introducing optimizers in Section 2.1.5.6. Assuming that we have some N-dimensional latent representation of the data, we would like to connect it to the final layer in the neural network, the output layer, which we explain next.

2.1.5.3 Output layer

The last layer in the ANN is the *output layer* $\{y_j\} = \{y_1, y_2, \ldots, y_M\}$, which we connect to the last hidden layer of the network. Similar to the hidden layer, we connect each unit in the last hidden layer to each unit in the output layer. We illustrate an example of this in Figure 2.21, where we see a complete, multilayered neural network (MLNN).

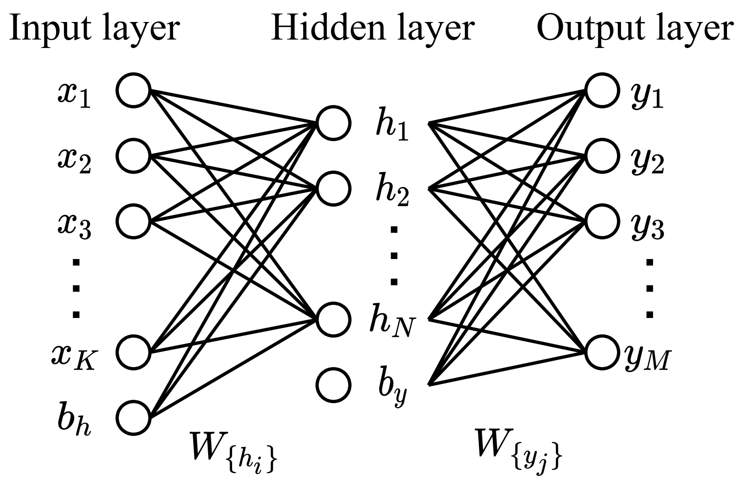

Figure 2.21: Multilayered neural network with one input, hidden and output layer.

Following, we define the output layer $\{y_j\}$ using a $M \times N$ weight matrix $W_{\{y_j\}}$, an M-dimensional bias vector b_y and an activation function $f_{\{y_j\}}$. More formally, we define the output layer $\{y_j\}$ as

$$\{y_j\} = f_{\{y_j\}}(W_{\{y_j\}} \cdot \{h_i\} + b_y). \tag{2.22}$$

We have now covered the different layers in an MLNN, but have yet to cover the different choices of activation function f in the neural network and how the MLNN learns. In the following sub-subsections, we look at choices of activation and loss functions, as well as optimizers for learning in the network.

2.1.5.4 Activation functions

The input data we use to feed into an ANN can contain complex patterns and have non-linear relationships. To learn such patterns and relationships, we apply an activation function to each layer before the result is sent to the proceeding layer. There are several choices of activation functions and following, we explain some of the most common ones.

The simplest type of activation is the identity function, which we define as

$$f(x) = x. \tag{2.23}$$

We commonly use the identity function when we want to pass on the values from one layer to another without modifying the value itself, as we use in the input layer from Section 2.1.5.1. Other choices of activation functions include *sigmoid*, *tanh*, *Rectified Linear Unit* (ReLU) and *softmax*. We visualize activation functions in Figure 2.22 and define them formally below as

$$f(x) = \frac{1}{1 + \exp(-x)} \text{ (sigmoid function)}, \tag{2.24}$$

$$f(x) = \frac{\exp(2x) - 1}{\exp(2x) + 1} \text{ (tanh function)}, \tag{2.25}$$

$$f(x) = \max\{x, 0\} \text{ (ReLU function)}, \tag{2.26}$$

and

$$f(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{K} \exp(x_j)}\ ,\ i \in \{1, \ldots, K\} \text{ (softmax function)}, \tag{2.27}$$

where K is the number of output values for the softmax layer.

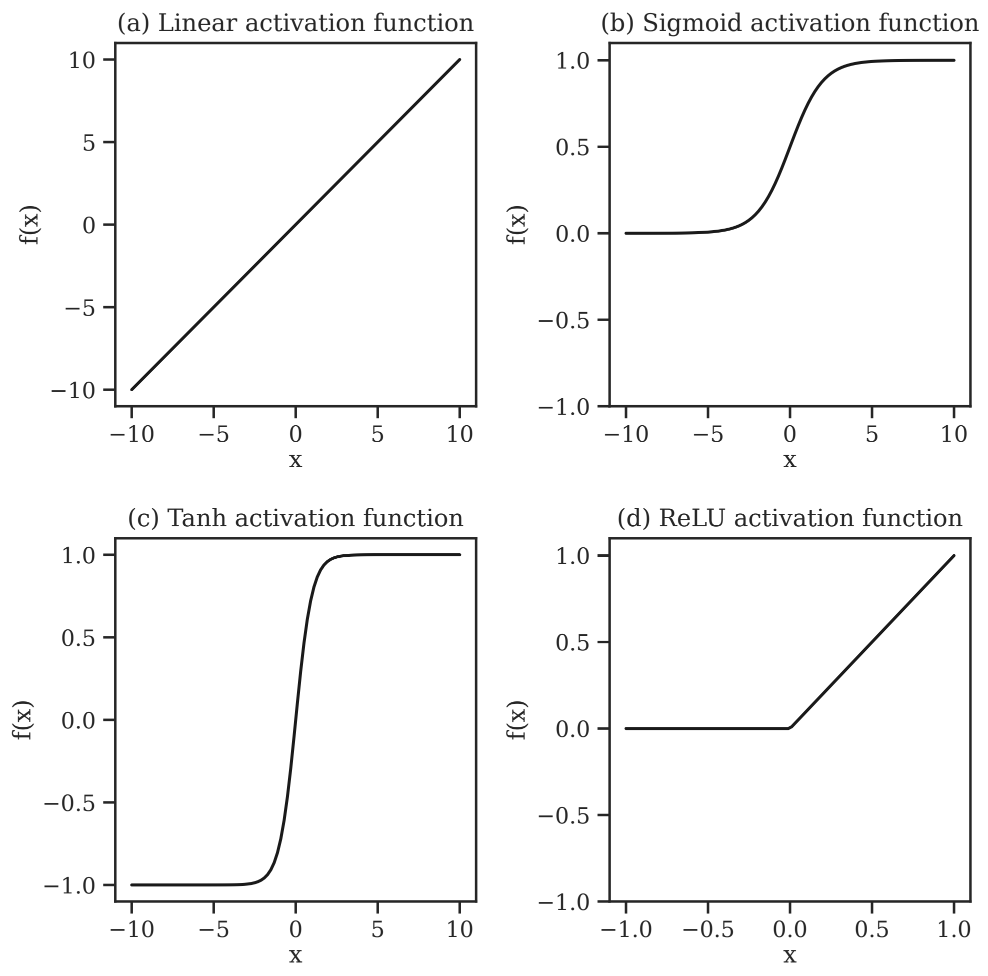

Figure 2.22: Four plots of various activation functions showing how they respond to different values of x.

The sigmoid and tanh activation functions were typically used in the early development of neural networks. The sigmoid activation function maps a value to a value in $(0, 1)$ and is particularly useful since it creates a probabilistic output. The tanh activation function has a similar shape to the sigmoid activation function and maps values to values in $[-1, 1]$. We show the relationship between the tanh and sigmoid activation functions as

$$\tanh(x) = 2 \cdot \text{sigmoid}(2x) - 1, \tag{2.28}$$

where we see that the tanh activation function is a rescaled version of the sigmoid activation function.

In more recent years, the ReLU activation function has become more popular, partly due to its computational simplicity. Both the sigmoid and tanh activation function suffers from *vanishing gradients* (i.e. gradients become zero, leading to practically no learning) and we typically use ReLU as a substitute to overcome this problem. This does not mean, however, that we can use the ReLU activation function without problems, as it can "die out" since it is non-differentiable at 0.

Up until now, we have only explained activation functions in the context of a single output value. To perform classification with K outputs, we typically use the softmax activation function, which we define in Equation (2.27). We interpret the output of the softmax activation function as the probabilities of the K outputs. Next, we look at loss functions in neural networks.

2.1.5.5 Loss functions

By connecting the different layers in an ANN, we have seen how we use some input data, send it through some hidden layer, and finally, get the predicted output values from the output layer. We denote the predicted output values as $\hat{y}$ and assume that we have the true output values as well, which we denote y. The *loss function* measures how much the predicted values $\hat{y}$ deviate from the true values y, and as such, the goal is to minimize this value. The output layer can have one or many outputs, and depending on the configuration of the network, the loss function changes as well. We separate the output types of an ANN into two categories, namely the *regression* and *classification* outputs.

In a regression type output, we usually predict some real-value quantity, such as height, weight or distance. For such problems, it is common to use the *mean squared error* (MSE) as the loss function. We calculate the MSE as the mean of the squared differences between the predicted and true values. More formally, we define MSE as

$$\mathrm{MSE}(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2, \tag{2.29}$$

where N is the length of y and $\hat{y}$.

For classification type outputs, we want to classify whether or not some input data belongs to two (binary, e.g. on or off, blue or red) or more classes (categorical, e.g. different types of animals). In a binary classification type output, we use the sigmoid activation function in conjunction with the *binary cross-entropy* (BCE) loss function. We define the binary cross-entropy loss function as

$$\mathrm{BCE}(\hat{y}, y) = -\left(y \cdot \log\left(\hat{y}\right) + (1 - y) \cdot \log\left(1 - \hat{y}\right)\right). \tag{2.30}$$

As opposed to the binary cross-entropy function, we use the *categorical cross-entropy* (CCE) function to compute the loss for multi-class classification output. We commonly use the softmax activation function in the output layer to create a multi-class probability distribution. Furthermore, we define the CCE loss function as

$$\mathrm{CCE}(\hat{y}, y) = -\sum_{c=1}^{C} y_c \cdot \log\left(\hat{y_c}\right), \tag{2.31}$$

where C is the number of classes in the multi-class classification output. In Equation (2.31), we observe that CCE is simply a generalization of the BCE loss function for multiple classes.

2.1.5.6 Optimizers

In this sub-subsection, we explain how an ANN can effectively make predictions from input data by learning its internal weights. In particular, we explain the gradient descent algorithm and how ANNs exploit it to perform efficient training of its internal parameters.

So far, we have discussed what we call the *forward pass* (or phase). A forward pass is simply the journey of the input data to the output layers where we in each step compute the output values at each layer and local derivatives using the current set of weights. Once we are at the output layer, the forward pass is complete and the *backward pass* commences. Recall that the objective of the neural network is to minimize the loss function. To do so, we compute the derivative of the loss function with respect to the weights in the input layer, using the chain rule from calculus. The derivative of the loss function tells the neural network which direction it should move each weight in to minimize the loss (i.e. in the negative direction of the derivative). To give an example of forward and backward passes in an ANN, consider the example in Figure 2.23.

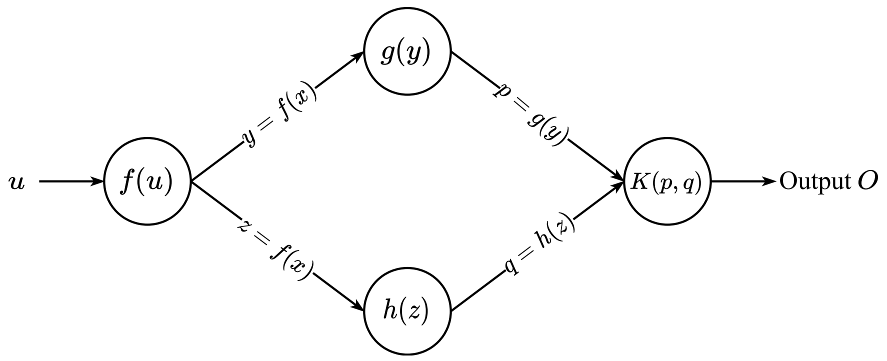

Figure 2.23: A neural network with one input node, one hidden layer with two nodes and one output layer.

In Figure 2.23, we have a network with one input node (neuron), a hidden layer with two nodes and a single output layer with one node. We denote the input to the network as u, which is the product of the input data and the weights in the input layer. For the activation functions of the network, we denote them as f, g and h. Moreover, we denote the results of the activation functions as y, z, p, q, and the function K combines the result of p and q resulting in the output value O. We assume that we apply the weights of the hidden layer to the previous layer's output values during g and h and that we apply the weights of the output layer at K. The forward pass in this network is straightforward: We start the input node, pass the data on to g and h and combine the results in the output layer. Now, for the backwards pass, we first compute the loss at the output node using a loss function L. Furthermore, we compute the derivative of L with respect to the input u. More formally, we compute

$$\begin{aligned}
\frac{\partial L}{\partial u} &= \frac{\partial L}{\partial O} \cdot \frac{\partial O}{\partial u} \\
&= \frac{\partial L}{\partial O} \cdot \left[\frac{\partial O}{\partial p} \cdot \frac{\partial p}{\partial u} + \frac{\partial O}{\partial q} \cdot \frac{\partial q}{\partial u} \right] \text{(chain rule)} \\
&= \frac{\partial L}{\partial O} \cdot \left[\frac{\partial O}{\partial p} \cdot \frac{\partial p}{\partial y} \cdot \frac{\partial y}{\partial u} + \frac{\partial O}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial u} \right] \text{(chain rule)} \\
&= \frac{\partial L}{\partial O} \cdot \left[\underbrace{\frac{\partial K(p,q)}{\partial p} \cdot g'(y) \cdot f'(u)}_{\text{Path on top}} + \underbrace{\frac{\partial K(p,q)}{\partial q} \cdot h'(z) \cdot f'(u)}_{\text{Path on bottom}} \right].
\end{aligned} \tag{2.32}$$

In Equation (2.32), we see how to calculate the derivatives for a relatively small neural network. There exists, effective and more general frameworks to derive the derivative of the loss with respect to the input values, but we leave those details out and kindly refer the reader to [Aggarwal, 2018, Chapter 1.3] for more details.

We have now gone over the forward and backward passes, which are the first two steps of the so-called *backpropagation* algorithm. The remaining step of the algorithm is to use the computed derivatives to update the weights of the ANN. To do this, we use the *gradient descent* (GD) algorithm. The main idea of gradient descent is to update the weights iteratively by moving them in the opposite direction of the gradient (i.e. the steepest descent) of the loss with respect to the weights. By doing so, it leads to better-fitting weights for the input-output data. In standard gradient descent, we perform its steps by

$$W \Leftarrow W - \alpha \cdot \frac{\partial L}{\partial W}, \tag{2.33}$$

where $W = (w_1, w_2, \ldots, w_d)$ is the matrix consisting of the d weights of an ANN. The learning rate α is a hyperparameter and determines how much learning we want to do in each step. The learning rate is usually set to a relatively low value, in the order of 10^{-2} to 10^{-5}. We illustrate the effect of the gradient descent algorithm with a small example in Figure 2.24, where we compute gradient descent for the paraboloid $f(x, y) = (x - 2)^2 + (y - 4)^2$. We set the starting point to be (-5, -10), the learning rate α to 0.05 and use 100 iterations. In Figure 2.24, we see that the gradient descent algorithms finds the minimum of $f(x, y)$ relatively fast.

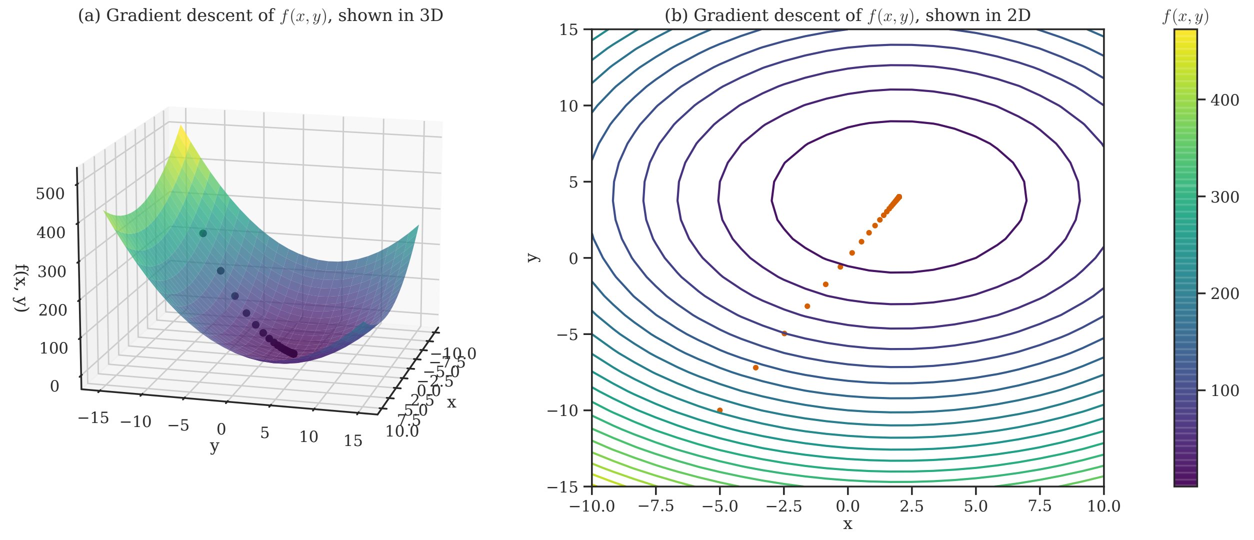

Figure 2.24: Gradient descent of the function $f(x,y) = (x-2)^2 + (y-4)^2$. We start at point (-5, -10), use a learning rate α of 0.05 and compute for 100 iterations.

Even though gradient descent works great for small applications, once we scale the number of parameters (weights) in the ANN to the more extremes (e.g. in the order of millions), it becomes impractical to compute for the entire training data at once. Furthermore, we observe that a loss function L usually can be written as a sum of the loss of the individual training data points, where L_i is the loss for training data point i. Thus, we define the individual loss L_i as

$$L = \sum_{i=1}^{n} L_i. \tag{2.34}$$

Using the observation that we are able to write the loss function as the sum in Equation (2.34), we introduce the *stochastic gradient descent* (SGD) method. Instead of performing gradient descent on the whole input data, SGD performs gradient descent for each input data separately. More formally, SGD performs its steps by

$$W \Leftarrow W - \alpha \cdot \frac{\partial L_i}{\partial W}, \tag{2.35}$$

where n is the number of input data points and L_i represents the loss of the ith input. We call SGD stochastic because a random sample of the training data is chosen for each iteration. An advantage that SGD has over GD, is that it runs a lot faster, at the expense of greater loss. Thankfully, there exists a variant of SGD which seeks to find a balance between speed and loss, namely the *mini-batch stochastic gradient descent* (mini-batch SGD) method. In mini-batch SGD, we use a batch $B = \{j_1, \ldots, j_m\}$ of input training data indices when computing the weight updating. In other words, mini-batch SGD performs its steps by

$$W \Leftarrow W - \alpha \cdot \sum_{j \in B} \frac{\partial L_j}{\partial W}. \tag{2.36}$$

Mini-batch SGD often finds the best trade-off between stability, speed and memory requirements. However, we note that the memory requirement increases with the use of mini-batches. This is because we have to store bigger matrices in memory during training. We typically choose batch sizes that are a power of 2 (e.g. 32, 64, 128 or 256), as most modern hardware architectures optimize for such values.

In addition to the different variants of gradient descent, there exist a bunch of other variants which can solve issues such as getting stuck in local minima or speeding up the training process. We leave out the technical details of all these strategies here, but kindly refer the reader to [Aggarwal, 2018, Chapter 3.5] for more details.

2.1.6 Intrinsic dimension estimation

In machine learning, the *manifold hypothesis* states that, in general, real-life high-dimensional data tends to live on a low-dimensional submanifold embedded within the high-dimensional space [Bengio et al., 2014, p. 16]. To understand more about the underlying structure of the data we are working with, we can estimate the dimension of the low-dimensional submanifold. We call the process of estimating the dimension of the low-dimensional submanifold *intrinsic dimension (ID) estimation.* More generally, a D-dimensional data set $X \in \mathbb{R}^D$ is said to have an ID equal to d if X lies entirely within a d-dimensional subspace of $\mathbb{R}^D$ [Lee et al., 2015]. We separate between *global* and *local* ID estimation methods, where global methods estimate the ID for the entire data set, and local methods estimate ID for each data point in the data set. We note, however, that it is fully possible and common in the literature to approximate local ID estimates by computing global ID estimates of a k-nearest neighbourhood around each data point $x \in X$. In the following sub-subsections, we introduce five methods for estimating the ID. We leave out the technical details of the ID estimation methods, as they are not the focus of this thesis. Instead, we give an overall explanation for each of the methods, and kindly refer the reader to the source of each method for more details. For each of the methods we describe below, we assume that we have some data set $X \in \mathbb{R}^{N \times D}$ consisting of N samples, where each sample is a D-dimensional vector. We base this subsection on [Lee et al., 2015] if we do not state otherwise. Following, our interest is mainly in local ID estimation methods for this thesis, as we will use them for the analysis of word embeddings (Section 2.2) in Section 3.3.3 and for polysemous word prediction in Section 3.3.4.

2.1.6.1 LPCA

One of the first and most simple kinds of ID estimation algorithm bases itself on the PCA algorithm (see Section 2.1.4.1). In particular, the ID estimation algorithm uses information from the eigenvalue decomposition of PCA to estimate the intrinsic dimension. We refer to this ID estimation algorithm as the *local PCA* (LPCA) method, which was first introduced in [Fukunaga and Olsen, 1971]. We refer to [Fukunaga and Olsen, 1971] when explaining the LPCA method. The LPCA method works as follows. First, we reduce the dimensionality of the original data set X from D dimensions to $\hat{d}$ dimensions, by applying PCA and making sure that most of the variance in the $\hat{d}$ dimensions are kept. To select $\hat{d}$, LPCA counts the number of eigenvalues that are greater than a portion α of the largest eigenvalue, from PCAs eigenvalue decomposition. A typical value for α is 0.1, meaning that $\hat{d}$ is set to the number of eigenvalues that are greater than 90% of the largest eigenvalue. The value $\hat{d}$ is then the estimated ID. The LPCA algorithm for estimating ID is a local estimator, meaning that estimates the ID for every data point $x \in X$. We typically implement LPCA using k-nearest neighbour, meaning that for every data point $x \in X$, we compute PCA of the k-nearest neighbourhood around x to estimate its ID.

2.1.6.2 KNN

A popular approach to estimate the ID is to use k-nearest neighbour (k-NN) graphs. In this sub-subsection, we explain the *k-NN algorithm* for ID estimation, as explained by [Carter et al., 2010, p. 651]. The k-NN algorithm uses the notion of the total edge length of the k-NN graph built on the data X, to estimate the ID $\hat{d}$. Let $D(x_i, x_j)$ denote the distance between two data points $x_i, x_j \in X$, and let $\mathcal{N}_{k,i}$ denote the set of the k-nearest neighbours of data point x_i. Then, we define the total edge length of the k-NN graph as

$$L_{\gamma,k}(X) = \sum_{i=1}^{n} \sum_{y \in \mathcal{N}_{k,i}} D(x_i, x_j)^{\gamma}, \tag{2.37}$$

where $\gamma > 0$ is a constant, weighting the distances between the data points in the k-NN graph. If we let $\gamma > 1$, then we emphasize big distances between data points. The authors of [Carter et al., 2010, p. 651] then argue that, assuming the manifold hypothesis holds for the data X (i.e. that the data X can be fully described using a submanifold $X' \in \mathbb{R}^{N \times d}$), it is possible to estimate $\hat{d}$ by applying non-linear least squares to an approximation of the total edge length. The k-NN algorithm for estimating ID is a global estimator, meaning that estimates the ID of the whole data X. It is possible to make it local by applying the procedure we explained in the introduction to ID estimation in Section 2.1.6.

2.1.6.3 TWO-NN

Estimating the ID can be a hard task, especially if the underlying manifold of the data is twisted and curved. It is for this reason, the authors of [Facco et al., 2017] propose a two nearest neighbours estimator for ID estimation, which we refer to as the *TWO-NN* method. TWO-NN uses the first and the second nearest neighbour of each data point in X to estimate the ID. The authors of [Facco et al., 2017] show that by using the ratio of the distance to the second nearest neighbour to the first one to create a linear regression model which estimates the ID $\hat{d}$. Following, the authors claim that this minimality reduces the effect of complex manifolds, varying density and reduces the computational cost. The TWO-NN algorithm for estimating ID is a global estimator, meaning that estimates the ID of the whole data X. It is possible to make it local by applying the procedure we explained in the introduction to ID estimation in Section 2.1.6.

2.1.6.4 MLE

Maximum likelihood estimation (MLE) is a method for estimating the parameters of a probability distribution, by either maximizing a likelihood function. [Levina and Bickel, 2004] propose a method that uses MLE to estimate the ID of data points $x \in X$. The idea is to estimate the ID locally, by assuming that the density $f(x)$ is constant for a small sphere $S_x(R)$ of radius R around x. Then, they use a Poisson process to measure the rate of the counting process $N(t, x)$ which measures the number of points falling onto the sphere $S_x(R)$ at time t. Furthermore, they use MLE on the Poisson process to estimate the local ID $\hat{d}$ for point x. Note that in practice, it is common to use k-nearest neighbours instead of radius R to find neighbours of x. An extension of the MLE method for estimating local ID was proposed by [Haro et al., 2008], which makes the ID estimator more robust to noise.

2.1.6.5 TLE

Tight Local Intrinsic Dimensionality Estimator (TLE) [Amsaleg et al., 2019] is a method for estimating the local ID of data points $x \in X$. The authors of TLE claim that the method works well in tight localities, i.e. within neighbourhoods of small size (e.g. using k-nearest neighbour). By using distances from x to its k-nearest neighbours, TLE estimates the local ID $\hat{d}$. The authors of [Amsaleg et al., 2019] claim that TLE can achieve more accurate ID estimates within small neighbourhoods around x (i.e. for small values of k), which again can improve the quality of algorithms that depend on local ID estimates.

2.1.7 Regression analysis

Regression analysis is a set of methods from statistics that estimate relationships between a *dependent variable* and one or more *independent variables*. To give an example, a dependent variable could be "income" and independent variables could be "education" and "experience". Regression analysis could help us to understand how income is affected by education, for instance. In this subsection, we look at three regression methods in particular: linear regression, lasso regression and logistic regression. We refer to [James et al., 2013, Fox, 2015] when describing concepts from regression analysis. Furthermore, we will use the lasso and logistic regression methods for the prediction of polysemous words in Section 3.3.4.

2.1.7.1 Linear regression

Linear regression, as the name suggests, attempts to find linear relationships between variables. The simplest form of linear regression is between a dependent variable X and a single independent variable Y. First, we assume that there is an approximately linear relationship between X and Y. Then, we model the linear relationship as

$$Y \approx \beta_0 + \beta_1 X, \tag{2.38}$$

where the approximate equals sign "$\approx$" means that we are *regressing* Y onto X. The two constants β_0 and β_1 are unknown and represents the *intercept* and *slope* in terms of the linear model. We refer these two constants as the *model parameters*. Following, we use some training to compute estimates of the models parameters, $\hat{\beta}_0$ and $\hat{\beta}_1$. We use the *hat* symbol, $\hat{}$, to denote some estimated or predicted value. To predict future values for Y, we use the estimated parameters $\hat{\beta}_0$ and $\hat{\beta}_1$ by computing

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 X, \tag{2.39}$$

where $\hat{Y}$ is the predicted value of Y. To estimate the model parameters $\hat{\beta}_0$ and $\hat{\beta}_1$, we must use some training data. Now, we let $X = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ and $Y = (y_1, y_2, \ldots, y_n) \in \mathbb{R}^n$ represent our data as two n-dimensional vectors. In machine learning terms, we are in a supervised setting, since we know the true labels y before trying to predict them. After applying some linear algebra, we rewrite *Equation* (2.39) for a single data point i as

$$\hat{y}_i = \begin{bmatrix} 1 & x_i \end{bmatrix} \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{bmatrix}, \tag{2.40}$$

where $\hat{y}_i$ is the predicted value for x_i. Since we have n values for X and Y, we need n equations similar to Equation (2.40), which we combine into a single matrix equation as

$$\hat{Y} = \underbrace{\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}}_{X'} \underbrace{\begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{bmatrix}}_{\hat{\beta}}, \tag{2.41}$$

where X' is the *model matrix*, consisting of ones in the first column and X in the second column, and $\hat{\beta} = \left(\hat{\beta}_0, \hat{\beta}_1\right)$ is a vector consisting of the models parameters. Using the *ordinary least squares* (OLS) method [Fox, 2015, p. 208], we get a closed-form expression for estimating the parameters $\hat{\beta}$, by computing

$$\hat{\beta} = \left(X'^{\top} X'\right)^{-1} X'^{\top} Y. \tag{2.42}$$

The OLS method minimizes the *residual sum of squares* (RSS), i.e. the sum of the squared differences between the true value y and predicted value $\hat{y}$. More formally, we define the objective of OLS as

$$\mathrm{RSS} = \sum_{i=1}^{n} \left(y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_i)\right)^2. \tag{2.43}$$

We illustrate the use of OLS in Figure 2.25, where see a clear relationship between the variables X and Y.

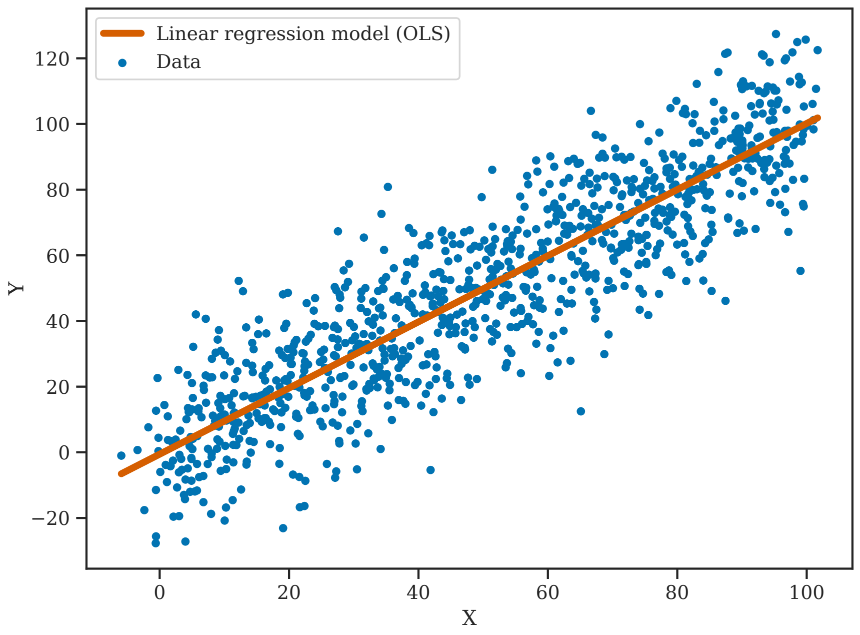

Figure 2.25: A linear regression model using OLS, with one dependent variable Y and one independent variable X. The red line shows the OLS model.

So far, we have only looked at the simplest form of linear regression, namely only using a single independent variable X to predict a value for Y. To generalize the linear regression to k independent variables, we now assume that our training data has k variables (which we also refer to as *features* or *columns*), i.e. $X \in R^{n \times d}$. Finally, we add the new columns from X to the model matrix X', by extending Equation (2.41) as

$$\hat{Y} = \underbrace{\begin{bmatrix} 1 & x_{11} & \dots & x_{1k} \\ 1 & x_{21} & \dots & x_{2k} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \dots & x_{nk} \end{bmatrix}}_{X'} \underbrace{\begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \vdots \\ \hat{\beta}_k \end{bmatrix}}_{\hat{\beta}}. \tag{2.44}$$

To estimate the model parameters $\hat{\beta}$, we compute them using Equation (2.42). Furthermore, we generalize the minimization objective of OLS in Equation (2.43) to use k independent variables as well, by computing

$$\text{RSS} = \sum_{i=1}^{n} \left[y_i - \left(\hat{\beta}_0 + \sum_{j=1}^{k} \hat{\beta}_j x_{ij} \right) \right]^2. \tag{2.45}$$

2.1.7.2 Lasso regression

Linear regression suffers from the fact that it has to use all independent variables to predict a value for the dependent variable. Imagine that we gather some 5-dimensional data X, and want to predict some quantity y using X. We perform some analysis on the data and notice that two of the features of the data are plain noise and most likely does not help to predict y. *Lasso regression* is a slight modification of linear regression that helps us with this problem. By adding a *penalty term* on the model parameters in the objective of the linear regression, lasso regression can push certain features to 0, essentially ”removing” them from the model. If we have two similar features, the lasso is also able to ”nullify” one of them, since the first one ”explains” the second one. As a result, the models we get from lasso regression are generally much easier to interpret. What we have described here is also referred to as *feature selection*, where the model automatically selects which variables are useful for prediction.

Lasso regression minimizes the residual sum of squares (RSS) plus some constant λ times the ℓ_1-*norm* of the model parameters β. We define the minimization objective as

$$\sum_{i=1}^{n}\left[y_i-\left(\hat{\beta}_0+\sum_{j=1}^{k}\hat{\beta}_j x_{ij}\right)\right]^2+\lambda||\beta||_1=\text{RSS}+\lambda||\hat{\beta}||_1, \tag{2.46}$$

where $\lambda \geq 0$ is a *regularization* parameter. We denote the second term of Equation (2.46), $\lambda||\hat{\beta}||_1$, as the ℓ_1-penalty, which is small when $\hat{\beta}_0$, $\hat{\beta}_1$, $\ldots$, $\hat{\beta}_k$ are small. When $\lambda = 0$, the penalty term has no effect and lasso regression produces the same result as standard linear regression. As $\lambda \to \infty$, the effect of the ℓ_1-penalty grows and the model parameters $\hat{\beta}$ approaches zero, and in some cases, some of the model parameters are be exactly zero. We define the ℓ_1-norm as $||\hat{\beta}||_1 = \sum|\hat{\beta}_j|$, where $|\cdot|$ is the absolute value.

2.1.7.3 Logistic regression

When describing linear regression models in Section 2.1.7.1, we assume that the response variable Y is *quantitative*. When working with *qualitative* data, on the other hand, linear regression fails to work. For example, colours are qualitative, taking on values such as blue, red, green or brown; we can not claim that red is greater than blue or brown is less than green (discarding RGB colour information). When we predict a qualitative response variable Y using some data X, we perform what we call a *classification* task. *Logistic regression* is a method that can predict the response variable Y when it falls into one of two categories (i.e. binary response). Examples of a binary response variable are "Yes"/"No", "Is a dog"/"Is not a dog" and the typical 0/1, which we commonly use in machine learning. Logistic regression creates a model that models the probability that Y falls belong to a particular category. In Figure 2.26, we further motivate the use of logistic regression for binary-response tasks and we see that linear regression is not well-suited for predicting values between 0 and 1.

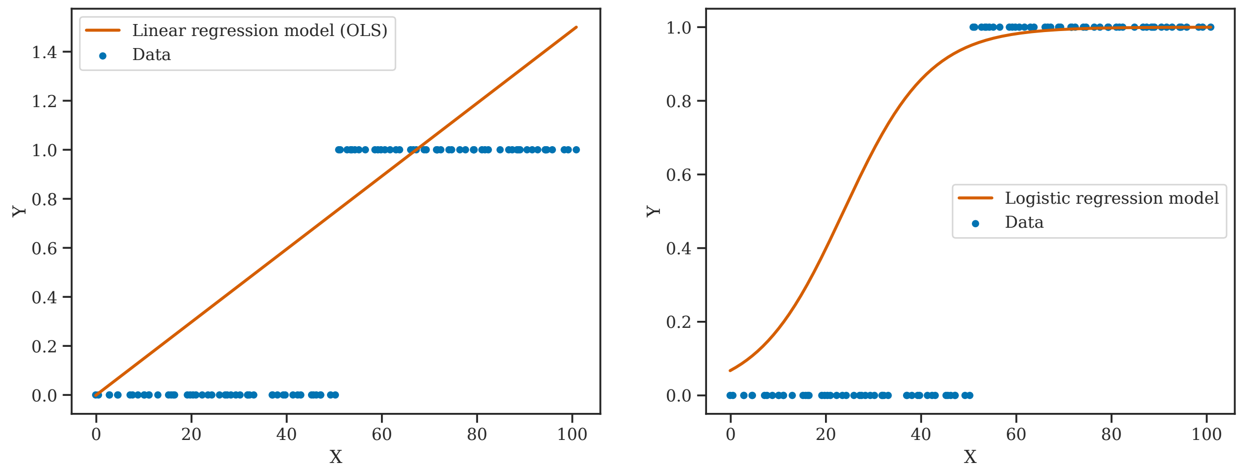

Figure 2.26: Illustration of linear and logistic regression models predicting some binary response variable Y on data X. We see that logistic regression is particularly well-suited for predicting values for Y between 0 and 1. Linear regression fails to model this relationship and even goes out of bounds for $X > 70$.

Let $p(X) = Pr(Y = 1|X)$ denote the probability of Y being 1 when we have the data X. If we want to model this relationship, we could use linear regression by

$$p(X) = \beta_0 + \beta_1 X. \tag{2.47}$$

Unfortunately, as mentioned earlier, linear regression is not a good fit for such problems and we would have the situation on the left of Figure 2.26. In logistic regression, the *logistic function* is used to ensure that the output value falls between 0 and 1, and we define it as

$$p(X) = \frac{\exp(\beta_0 + \beta_1 X)}{1 + \exp(\beta_0 + \beta_1 X)}. \tag{2.48}$$

The plot to the right of Figure 2.26 shows the effect of the logistic function, creating an S-shaped like curve that, for high values of X, creates values close to, but never greater than 1. On the other hand, the logistic function creates values close to, but lever less than 0 for low values of X. Manipulating Equation (2.48) a bit, we get that

$$\frac{p(X)}{1 - p(X)} = \exp(\beta_0 + \beta_1 X), \tag{2.49}$$

where we refer the left-hand-side of Equation (2.49), i.e. $p(X)/[1 - p(X)]$, as the *odds*. As $p(X)$ increases, the odds increases exponentially towards ∞. Taking the logarithm of both sides of Equation (2.49), we finally arrive at

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X, \tag{2.50}$$

where we refer the left-hand-side of Equation (2.50) as the *logit*. Thus, we see that the logistic regression model has a logit that is linear in X. If we increase X by one unit, then the log odds increase by β_1. However, note that the relationship between $p(X)$ and X in Equation (2.48) is not a straight line. For this reason, the amount of $p(X)$ changes due to a single unit change in X depends on the current value of X. In Figure 2.26, we see that once X reaches a certain threshold (e.g. $X = 30$), the rate at which Y changes decreases towards zero.

If we would like to model logistic regression using multiple predictors, we only have to perform a simple extension from simple to multiple linear regression. We generalize Equation (2.50) as

$$\begin{aligned}\log\left(\frac{p(X)}{1-p(X)}\right) &= \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_k X_k \\ &= \beta_0 + \sum_{i=1}^{k} \beta_i X_i,\end{aligned} \tag{2.51}$$

where $X = (X_1, X_2, \ldots, X_k)$ are k predictors.

To estimate the parameters $\beta_0, \beta_1, \ldots, \beta_k$ for k predictors, it is common to use the *maximum likelihood* method. We do not go into detail about the maximum likelihood method here, but kindly refer the reader to [Fox, 2015, p. 214] for more details.

2.1.8 Model selection

Model selection is an important aspect of modern machine learning. When working with machine learning, we would like to understand which model solves the problem the best, and model selection helps us with this. Model selection is the task of selecting a model for a particular problem using the data at hand. We look at a couple of methods for performing model selection, namely using train/validation/test splits and K-fold cross-validation. We refer to [James et al., 2013] when describing model selection methods. Furthermore, we will use model selection when training lasso and logistic regression model for supervised word polysemy prediction in Section 3.3.4.

2.1.8.1 Train, validation and test splits

When training a supervised machine learning model, we typically have some data X and corresponding labels y. The task is to predict the labels y using the data X. Since we do not know apriori which model or model parameters to use, the most common way to figure this out is by performing model selection. The simplest kind of model selection for machine learning models is to split the data X and labels y into three data sets, namely the *train-*, *validation-*, and *test data sets*. The new data sets are chosen at random and do not overlap. An example of a train/validation/test split ratio could be 80/10/10, where we use 80% of the data in the training data set, 10% of the data in the validation data set and 10% of the data in the test data set. Exactly how we split the data sets into the smaller train/validation/test data sets depends on the application and how much data we are working with. In more modern machine learning models, e.g. using artificial neural networks, it is common to use up to 99% of the data for training, as long as we have a big enough data set (e.g. > 1 million data points).

We use the train data set to learn the models' parameters from the data, e.g. a linear regression models β parameter. The train data set is usually much larger than the validation or test data sets. To evaluate the parameters of the model, we use the *validation data set*. When we evaluate a trained model using a validation data set, we perform what we call the *validation set approach*. The validation data set provides an unbiased estimate of a models' fit on the training data set while tuning some *hyperparameter*. A hyperparameter is a parameter that we choose for the model beforehand, in contrast to a model parameter, which the model learns internally. Tuning hyperparameters can be a difficult task, especially if we have a lot of hyperparameters with various values. A typical way of performing hyperparameter optimization is by using *grid search*, which tries out all combinations of all hyperparameter values. Grid search can be computationally expensive, especially if we would like to try out many choices of hyperparameters. For this reason, we typically use grid searches to find potential ranges where the optimal hyperparameters live and then narrow down the search for smaller ranges of hyperparameters to find more optimal ones.

After selecting a model or a set of model parameters using the validation data set, we use the test data set to give an unbiased estimate of how the model performs on unseen data. Additionally, we refer to the test data set as the *hold-out data set*, because its only use is at the end of model selection. Note that we do not necessarily require to have a test data set available to perform model selection, as it only evaluates the performance of the final model. A common approach is also to exclude the test data set and instead split the original data X into training and validation only. By doing so, we are unable to get an unbiased estimate of the models' performance on unseen data. We illustrate the process of splitting data into train/validation/test splits in Figure 2.27.

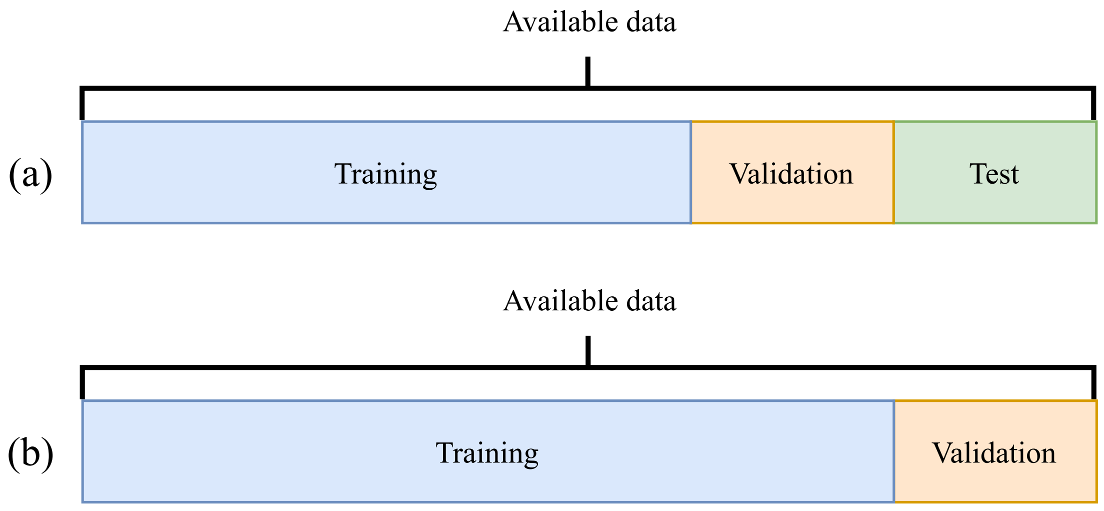

Figure 2.27: Example of ways to partition a data set for model selection. The partition on the top (a) splits the data into training, validation and test, while the partition on the bottom (b) splits the data into training and validation only.

The validation set approach is simple and works fine in practice. There are, however, a couple of drawbacks when using this method. Depending on which observations we include in the training and validation data sets, the validation estimate of the error can vary a lot. In addition to this, since the validation set approach only uses a subset of the observations to train the model, and the fact that statistical models perform worse when we train them on less data, the validation error rate tends to overestimate the test error rate of a model trained on the entire data set. In the next sub-subsection, we introduce k-fold cross-validation, an improvement over the validation set approach that addresses these two drawbacks.

2.1.8.2 k-fold cross-validation

An alternative to the validation set approach is the *k-fold cross validation* (CV) method. The k-fold CV method randomly divides the training data X into k groups, or *folds*, of approximately equal size. We treat the first fold as the validation data set and train the model on the remaining $(k-1)$ folds. We compute the validation error Err_1 on the first validation data set. Following, this procedure is repeated k times, and for each time, we use a different group of data points from the original training data X as the validation data. This results in k estimates of the test error, i.e. Err_1, Err_2, ..., Err_k. We compute the total k-fold CV error estimate by taking the mean of these values, as

$$\text{CV}_k = \frac{1}{k}\sum_{i=1}^{k} \text{Err}_i. \tag{2.52}$$

Choosing a value for k is a hard problem. The most common choice is to set $k = 5$ or $k = 10$, depending on the problem. We kindly refer the reader to [James et al., 2013, Section 5.1.4] for more details on choosing a value for k.

We show an illustrative example of k-fold CV in Figure 2.28, where we see a typical set-up when using a k-fold CV, namely to split all available data into training and test data sets. During the training process, the training data set is furthermore split into training and validation data sets for that particular k.

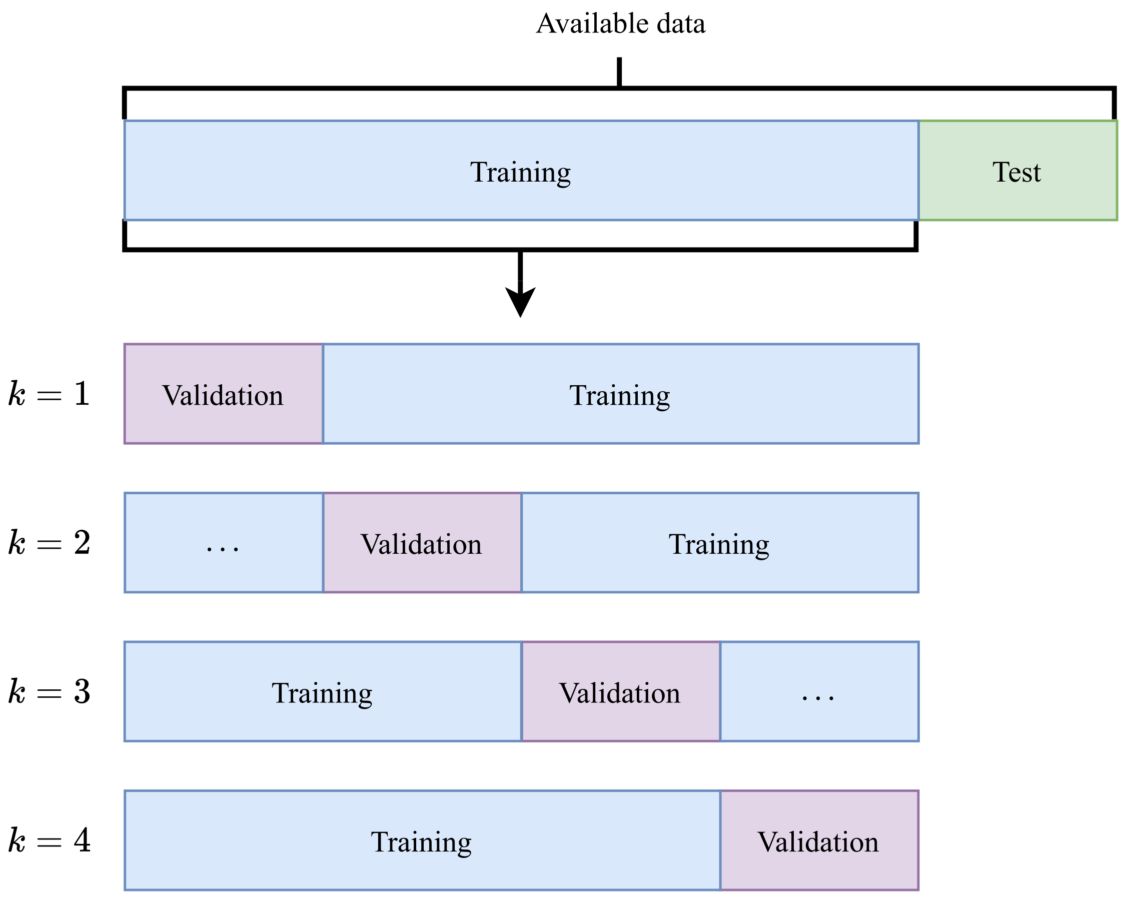

Figure 2.28: Data split into training and test data sets. To perform model selection, we use k-fold cross-validation. In this example, we have a 4-fold cross-validation setting. As the k increases, we have different data sets for both training and validation.

The benefit of using k-fold CV is that, by both training on different subsets of the training data and evaluating the model on different validation data sets, the estimated test error becomes more accurate and we get less varying results, as long as we select a good value for k.

2.1.9 Performance metrics

In this subsection, we introduce some common choices of performance metrics for regression and classification problems. We use performance metrics to evaluate machine learning models on data sets. In the following sub-subsections, we look at classification accuracy, the confusion matrix and sensitivity. Furthermore, we will use the performance metrics below to evaluate the word2vec model in Section 3.1.4 and for evaluating results from prediction of polysemous words in Section 3.3.4.

2.1.9.1 Classification accuracy

Classification accuracy (which we also refer to as *accuracy*) is the ratio between correct predictions to the total number of samples. We typically use classification accuracy in classification problems, as it is a simple and interpretable metric. Let C_p be the number of correct predictions and let N be the total number of samples. We compute classification accuracy as

$$\mathrm{ACC}(C_p, N) = \frac{C_p}{N}. \tag{2.53}$$

One pitfall of using classification accuracy is when we are dealing with classification problems with unbalanced data sets. Imagine that we want to compute the performance of a classification model for classifying whether or not a patient has a rare but fatal disease. If we use classification accuracy, it is not uncommon to see high accuracies (e.g. 99.9%), when in reality, the number of correct classifications for whether or not a patient has the disease is significantly lower. In other words, we get a false sense of the models' performance. To deal with such cases, it is more common to use metrics such as sensitivity (see Section 2.1.9.3), which deals with class imbalance much better and is well-suited for specific tasks.

2.1.9.2 Confusion matrix

Confusion matrices explains the performance of classification models by creating a matrix of predicted values to true labels. We see such a confusion matrix in Figure 2.29, where we have a binary classification problem (two classes; 0 and 1). As we see in Figure 2.29, we have four terms which describe the performance of the model, namely the *True Negative* (TN), *False Negative* (FN), *False Positive* (FP) and *True Positive* (TP) terms. TN is the number of predicted negative classes when the true classes were negative, while on the other hand, TP is the number of predicted positive classes when the true classes were positive. Off the diagonals of Figure 2.29, we see FP, which is the number of predicted positive classes when

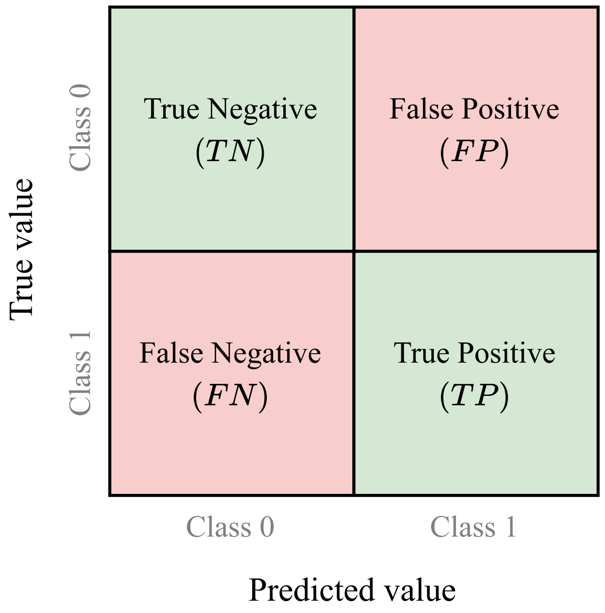

Figure 2.29: A typical example of a confusion matrix of a binary classification problem, with classes 0 and 1.

the true classes were negative, and FN, which is the number of predicted negative classes when the true classes were positive. When computing accuracy we would like, ideally, the number of TN and TP samples to be as high as possible and the number of FN and FP samples to be as low as possible. In cases where we have a high class imbalance (e.g. from example in Section 2.1.9.1), it is more common to optimize either TN or TP to be as high as possible, effectively minimizing either FN or FP. We look at one such metric, namely sensitivity in Section 2.1.9.3.

2.1.9.3 Sensitivity

Sensitivity is a performance metric which measures the ability of a classification model to correctly classify a positive class (e.g Class 1 in Figure 2.29). The sensitivity only focuses on the values where the true class is positive, i.e FN and TP for a binary classification problem. More formally, we define sensitivity as the portion of correctly predicted positive classes to the number of samples with a positive class. We compute the sensitivity as

$$\text{SEN} = \frac{TP}{TP + FN}. \tag{2.54}$$

If we, on the other hand, would like to look at the ability of a classification model to correctly classify a negative class (e.g. Class 0 in Figure 2.29), we can, with minor modification of Equation (2.54), change to focus on the prediction of negative classes. This modification creates a measure commonly referred to as *specificity*.

2.2 Word embeddings

In this section, we will formulate the problem of creating efficient vectorized representations of text and explain methods for doing so. In particular, we will discuss ways of representing text numerically in Section 2.2.1, describe how we can create word embeddings and details around the word2vec method in Section 2.2.2, from architectural choices to presenting word2vec as an artificial neural network. Finally, we will introduce a couple of alternative models for learning word embeddings in Section 2.2.3 and explain how to evaluate word embedding models in Section 2.2.4.

2.2.1 Numerical representation of text

Machine learning methods take in vectors (arrays of numbers) as input. When we want to work with text, we have to come up with some procedure for converting text into a vector, i.e. vectorizing the text. In this subsection, we create unique representations for words in a text, discuss one-hot encoding of words and the motivation behind creating word embeddings.

2.2.1.1 Unique representation for each word

A first strategy for vectorizing text could be to assign a unique number for each word in the text. We use the same order as the words that appear in the text to assign a unique number. We call the set of unique words that appear in the text the *vocabulary* and denote it as V. Furthermore, we define the number of unique words in the text to be the *vocabulary size* and denote it as $|V|$. Finally, we replace each word in the text with its respective number. Let us consider an example of a (lower-cased) sentence, which we define as

$$S = \text{the cat sat on the mat.} \tag{2.55}$$

Following, we convert the words into numbers using the numerical order they appear in the text, e.g. the $\mapsto$ 0, cat $\mapsto$ 1, sat $\mapsto$ 2, etc. We convert the sentence in Equation (2.55) to numbers as

$$S = 0\ 1\ 2\ 3\ 0\ 4. \tag{2.56}$$

In Equation (2.56), we see a numerical representation of the original sentence in Equation (2.55) and may use it for machine learning modeling. However, there are some problems with this method:

- The encoding of words into number is arbitrary (does not capture any relationship between words)
- Machine learning models might learn some natural ordering of the encodings, which can lead to bad results during inference. This is because the encoding of the words does not capture the relationship between the words.

In the next sub-subsection, we look at another method for encoding words, using one-hot encodings. Furthermore, we apply it to the example sentence in Equation (2.55).

2.2.1.2 One-hot encoded words

One-hot encoding is a method for converting categorical data into numeric data. Essentially, we create a unique, sparse vector consisting of all zeros, except for the value at the index of the element of interest, which we set to one. For instance, if we have the words "north", "east", "south", "west", then their one-hot encodings could be

$$\text{north} \mapsto \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \text{east} \mapsto \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \text{south} \mapsto \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \text{west} \mapsto \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

Let the vocabulary $V = \{w_1, w_2, ..., w_{|V|}\}$ to be the set of unique words in a text. Then, we define the one-hot encoding of a word, e_{w_i}, to be the $|V|$-dimensional vector of all zeros, except for the value at index i which is one. Using the definition of one-hot encoding, we convert the words from the sentence in Equation (2.55). Precisely, we convert the sentence as

$$S = \begin{pmatrix}1\\0\\0\\0\\0\end{pmatrix} \begin{pmatrix}0\\1\\0\\0\\0\end{pmatrix} \begin{pmatrix}0\\0\\1\\0\\0\end{pmatrix} \begin{pmatrix}0\\0\\0\\1\\0\end{pmatrix} \begin{pmatrix}1\\0\\0\\0\\0\end{pmatrix} \begin{pmatrix}0\\0\\0\\0\\1\end{pmatrix}, \tag{2.57}$$

where we see that we discard the the ordinal relationship of the numerical representation in Equation (2.56). However, there are some downsides with this approach, as well:

- As with the numerical representation in Section 2.2.1.1, one-hot encoded vectors does not capture the relationship between words.
- One-hot encoded vectors are sparse (meaning, most values are zero). Imagine if we had 1000 words in the vocabulary, then one would create a vector consisting of 99.9% zeros. In practice, the vocabulary size is in the terms of 10^5 to 10^7 [Mikolov et al., 2013b], i.e. by using one-hot encoded vector representations we are extremely inefficient in terms of space. We note, however, that there exist efficient methods for dealing with sparse vectors.
- One-hot encoded vectors are high-dimensional (same as the number of words in the vocabulary, $|V|$).

In the next sub-subsection, we look at word embeddings, which solves many of the problems of the numerical representations presented so far.

2.2.1.3 Word embeddings

In comparison to the high-dimensional one-hot encoded vectors of words, *word embeddings* are low-dimensional dense vector representations. Word embedding models learn their word embeddings from the data (e.g. texts) directly, whereas, for one-hot encoded vectors, we can arbitrarily define them (i.e. its ordering may change). Also, due to the lower dimensionality of word embeddings, it has to pack more information about words into less space. Moreover, word embeddings use all their dimensions to learn hidden relations and concepts of words, in contrast to one-hot encodings, which effectively only uses a single dimension (e.g. the position of the word in the vocabulary). Common choices for the dimensionality of word embeddings range from 50 to 600 [Mikolov et al., 2013a], depending on the amount of training data. In the next subsection, we look at a classic family of methods for creating such word embeddings, namely word2vec.

2.2.2 Learning word embeddings

Word2vec was first introduced by Mikolov et al. in 2013 [Mikolov et al., 2013a]. It is a family of methods for learning dense and efficient vector representations of words. In the same year, Mikolov et al. published a follow-up paper, [Mikolov et al., 2013b], which includes several extensions that improve both the quality of the word embeddings and the training speed. In this subsection, we explain the details of the original word2vec paper, [Mikolov et al., 2013a], and the introduced extensions in the follow-up paper, [Mikolov et al., 2013b]. We base this subsection on [Mikolov et al., 2013a, Mikolov et al., 2013b]. Furthermore, we will use the word2vec method to create word embeddings for the analysis in Section 3.1.

2.2.2.1 Architectures

The authors of the word2vec paper introduced two models for learning distributed representations of words that try to minimize computational complexity, namely the *continuous bag-of-words model* (CBOW) and the *continuous Skip-gram model.* Both models achieve high quality (see Section 2.2.4 for evaluation of word2vec models) word embeddings and share some core idea as to how we can create good vector representations of words. In this sub-subsection, we go through both models and explain the similarities and differences.

Continuous bag-of-words model
The continuous bag-of-words model (CBOW) tries to predict a target word given some context words around it. Essentially, we select a target word w_t, where t is the index of the current target word in our training data, and a number C denoting the number of words to the left and the right of w_t. We also refer to C as the *window size.* Let $S = 2C$ denote the

total number of words to the left and right. By combining the context word embeddings, we predict the target word w_t. We illustrate the CBOW model in Figure 2.30.

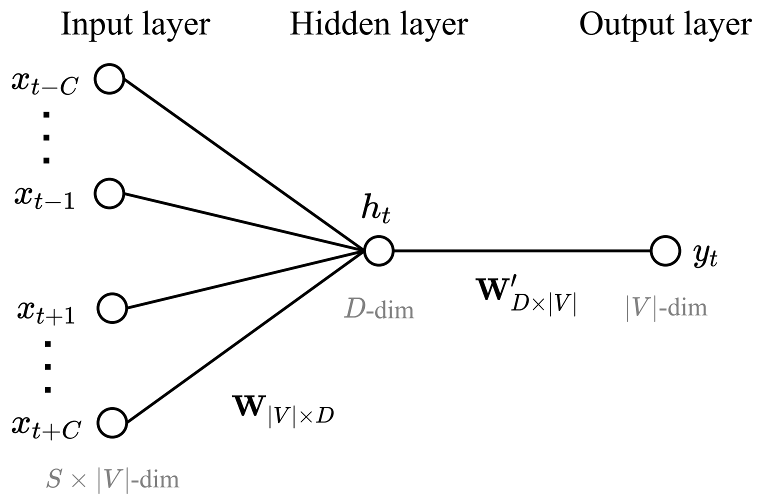

Figure 2.30: An illustration of the CBOW architecture. The input value x_{t+k} is the one-hot encoded vector of word w_{t+k}, where $k \in \{-C, \ldots, -1, 1, \ldots, C\}$, $t \in \{1, \ldots, T\}$ is training word position and C is the window size. The output of CBOW is a $|V|$-dimensional vector y_t of probabilities for sampling the target word w_t.

More formally, we are considering a sequence of T training words $w_1, w_2, \ldots, w_T$. The words w_t belong to some vocabulary V consisting of $|V|$ unique words, $1 \leq t \leq T$. The models task is to maximize the average log probability of sampling the word w_t, given the context words $w_{t-C}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+C}$. The objective of the CBOW model then becomes

$$\frac{1}{T} \sum_{t=1}^{T} \log p(w_t | w_{t-C}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+C}). \tag{2.58}$$

Through it is not clear from the original authors of word2vec [Mikolov et al., 2013a, Mikolov et al., 2013b], we typically use two weight matrices, W and W', when setting up the word2vec model [Rong, 2016]. The first weight matrix, W, is a $|V| \times D$ matrix, mapping the input words (we typically represent them using one-hot encodings) to their internal word embeddings, where $|V|$ is the vocabulary size, and D is the number of dimensions in the embedding layer. The second weight matrix, W', is a $D \times |V|$ matrix mapping from the embedding layer to the output prediction. In practice, we use the first weight matrix W as word embeddings, but we note that some models utilize both weight matrices (e.g. GloVe in Section 2.2.3.1).

Continuous Skip-gram model

The continuous Skip-gram model is similar to CBOW, and in fact, the Skip-gram model tries to do the opposite; instead of predicting a target word given some context words, it tries to predict context words given some target word. We note that the ordering of the predicted context words does not matter. We illustrate the Skip-gram model in Figure 2.31.

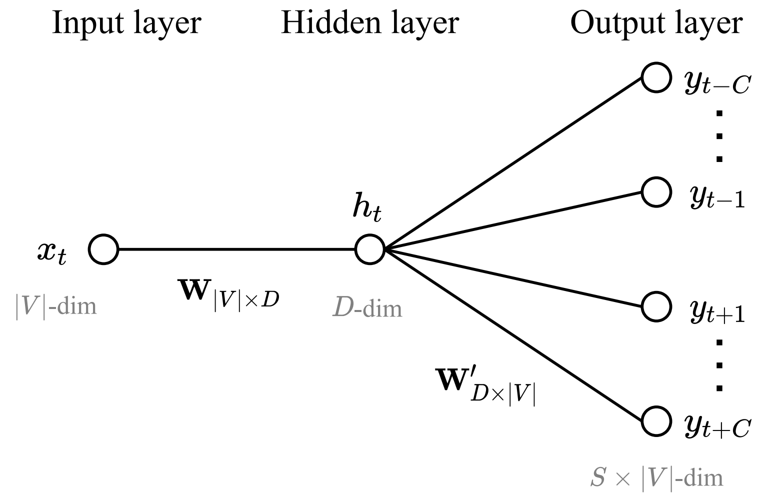

Figure 2.31: An illustration of the Skip-gram architecture. The input value x_t is the one-hot encoded vector of word w_t, where $t \in \{1, \ldots, T\}$ is training word position and C is the window size. The output values are $|V|$-dimensional vectors y_{t+k} of probabilities for sampling the word w_{t+k}, where $k \in \{-C, \ldots, -2, -1, 1, 2, \ldots, C\}$. We note that the ordering of the vectors from the output layers does not matter, as we only want to predict that a particular word belongs to its contextual words.

With the Skip-gram model, we also have some target word w_t and context words around it. Let C be the maximal distance from a target word to its contextual words. For each input to the model, we randomly sample a number R in the range $[1, C]$ and denote this as the *context size*. In other words, for each target word w_t we have R context words around it, $w_{t-R}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+R}$. The objective of the Skip-gram model then becomes

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-R\leq j\leq R, j\neq 0}\log p(w_{t+j}|w_t). \tag{2.59}$$

More generally, we define the objective of the Skip-gram model as such

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{w_O \in cw(w_I)} \log p(w_O|w_I), \tag{2.60}$$

where w_I is the input word (e.g. target word), w_O is the output word (e.g. context word) and $cw(w_I)$ is a function which returns the context words around input word w_I.

Similar to CBOW, the Skip-gram model uses the two matrices W and W' for mapping from input to embedding layer and embedding layer to output respectively. Mikolov et al. report that the Skip-gram model performs better than the CBOW model overall, shown by their experiments in [Mikolov et al., 2013a]. For this reason, and due to the scope of the thesis, we will focus on the Skip-gram model. For the rest of the thesis, we refer to the weight matrix W when talking about word embeddings of a word2vec model.

2.2.2.2 Negative Sampling

In the Skip-gram model, we typically compute $p(w_O|w_I)$ using the softmax function (see Equation (2.27)). More formally, we define it as

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^\top v_{w_I}\right)}{\exp\sum_{k=1}^{|V|}\left({v'_{w_k}}^\top v_{w_I}\right)}, \tag{2.61}$$

where v_w and v'_w are the "input" and "output" vector representations of the word w, and $|V|$ is the number of words in the vocabulary. There are some downsides with this formulation, however. In practice, it becomes hard to compute since the summation in the denominator of Equation (2.61) depends on the number of words in the vocabulary, which is often large ($10^5 - 10^7$ terms).

To deal with the computational requirements of the original Skip-gram model, [Mikolov et al., 2013b] first show that using *hierarchical softmax*, we end up with a viable strategy. Hierarchical softmax is an efficient way of computing the softmax function; instead of evaluating $|V|$ words to compute the probability in Equation (2.61), we only have to evaluate $\log(|V|)$ words.

As an alternative to hierarchical softmax, Mikolov et al. introduced *negative sampling* in [Mikolov et al., 2013b]. Negative sampling builds on the concept of distinguishing target words w_t from words randomly sampled from the vocabulary. In particular, we randomly sample words from the vocabulary using the *unigram distribution* raised to the power of α. The unigram distribution is a distribution used to sample random words from the vocabulary using the word occurrence counts. [Mikolov et al., 2013b] claim that by raising the unigram distribution to the power of $\alpha = 3/4$, we get the best result. Furthermore, we refer to this unigram distribution as the noise distribution $P_n(w)$. Note that we can also apply the negative sampling method to CBOW, but we leave these details out.

Before we can explain negative sampling, we define the positive and negative target-context pairs. Given a vocabulary V, a target word w_t and the target words contextual words $w_{t-R}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+R}$ for some window size R, we define a *positive target-context pair* to be the pair of the target word w_t and a contextual word w_{t+j}, $-R \leq j \leq R, j \neq 0$, i.e. the pair (w_t, w_{t+j}). Furthermore, we define a *negative target-context pair* as the pair of the target word w_t and a word w_r, i.e. the pair (w_t, w_r). We randomly sample the word w_r from the noise distribution $P_n(w)$.

To explain the idea of negative sampling, we look at how we compute the softmax loss, and in particular, how we only use a subset of all the words in the vocabulary. In particular, for each word in the text we are training on, we create a positive target-context pair (w_t, w_{t+j}), $-R \leq j \leq R, j \neq 0$. Furthermore, we generate k negative target-context pairs for each word, where k is in the range of $5-20$ for small training sets and $2-5$ for big training sets. We let $W_{np} = \{w_i | i \in 1, \ldots, k\}$ be the set of k negatively sampled words from the noise distribution $P_n(w)$. With these details in mind, the objective of negative sampling becomes [Mikolov et al., 2013b, Rong, 2016]

$$\log \sigma \left({v'_{w_O}}^\top v_{w_I} \right) + \sum_{w_i \in W_{np}} \log \sigma \left(-{v'_{w_i}}^\top v_{w_I} \right), \tag{2.62}$$

where σ is the sigmoid function (see Equation (2.24)), and v_t and v'_t are the "input" and "output" vector representations of the word w. Note that the objective in Equation (2.62) can be seen as a special case of the negative cross-entropy loss function. Furthermore, we replace every $\log p(w_O | w_I)$ in the original Skip-gram objective function of Equation (2.60) by the objective in Equation (2.62). As such, we define the objective of the Skip-gram model with negative sampling as

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{w_O \in cw(w_I)} \log\sigma\left({v'_{w_O}}^\top v_{w_I}\right) + \sum_{w_i \in W_{np}} \log\sigma\left(-{v'_{w_i}}^\top v_{w_I}\right). \tag{2.63}$$

In Equation (2.62), we see that we only have to compute for $(1 + k)$ words, which is a big improvement over computing for $|V|$ words (assuming that $|V|$ is much larger than k). [Mikolov et al., 2013b] also report that by using negative sampling, we increase the quality of the word embeddings.

2.2.2.3 Subsampling of words

When training a word2vec model, we typically have to train on big text corpora to achieve high-quality word embeddings. However, as the number of training words increases, the discrepancy between rare and frequent words increase as well. When using negative sampling, we are sampling negative target-context pairs from the vocabulary, which depends on the unigram distribution. In English text corpora, words such as "the", "of", "is" can easily occur hundreds of millions of times and usually provide less information than more rare words. For this reason, we apply a simple, yet efficient subsampling scheme to counter the imbalance between rare and frequent words; before we process the text corpora into target-context pairs, we discard each word w_t with a *discard probability* [Mikolov et al., 2013b, Levy et al., 2015]. We define the discard probability as

$$P_d(w_t) = 1 - \sqrt{\frac{t}{f(w_t)}}, \tag{2.64}$$

where $f(w_t)$ is the (relative) frequency of word w_t and t is a chosen threshold, usually around 10^{-5}.

We note, however, that in the original source code of word2vec [Mikolov, 2013c, line 407], they use a slightly modified formula. We define the modified formula as

$$P_d(w_t) = \frac{f(w_t) - t}{f(w_t)} - \sqrt{\frac{t}{f(w_t)}}. \tag{2.65}$$

Furthermore, we will use subsampling of words when implementing word2vec in Section 3.1, and in particular, we will use Equation (2.65) for computing the discard probability.

2.2.2.4 Learning word embeddings for phrases

Phrases are groups of words that occur frequently together, such as ”New York” or ”programming languages”. Naturally, we would like to combine such words into a single token, such that word embedding models can learn them. To learn word embeddings for phrases, [Mikolov et al., 2013b, p. 5-6] propose a simple, yet efficient data-driven approach, where they combine phrases using their word counts. In particular, we define the phrase score for two words w_i and w_j as

$$\text{score}(w_i, w_j) = \frac{\text{freq}(w_i, w_j) - \delta}{\text{freq}(w_i) \cdot \text{freq}(w_j)}, \tag{2.66}$$

where w_i and w_j are bigrams, or two neighbouring words, from the vocabulary, $freq()$ returns the how many times the word (or bigram) occurs in the vocabulary and δ is a hyperparameter for preventing long phrases consisting of infrequent words from appearing. We refer to this phrase-learning procedure as *word2phrase*, as specified by the original source code [Mikolov, 2013b]. If the score in Equation (2.66) is above the δ threshold hyperparameter, we accept the bigram into the vocabulary and we replace all occurrences of the two words w_i and w_j where they are next to each other. We use word2phrase to convert phrases into single words when preprocessing data for training our word2vec model in Section 3.1.1.

2.2.2.5 Word2vec as an artificial neural network

Typically in the literature, word2vec is presented using the Equations (2.58), (2.61) and (2.62). However, we explain how to set word2vec up as an artificial neural network (ANN, Section 2.1.5), in the sense that we implement it later in the thesis (see Section 3.1). In particular, we explain how to set up the Skip-gram model with negative sampling as an ANN. This ANN consists of three fully-connected layers of artificial neurons [Rong, 2016], and we illustrate it in Figure 2.32. The ANN we explain below acts as a baseline when implementing word2vec in Section 3.1.2.

Input layers
We have two input layers in our ANN; one for the target word w_t and one for the context word w_{t+j}, $-R \leq j \leq R, j \neq 0$. The values to the input layers are typically one-hot encoded

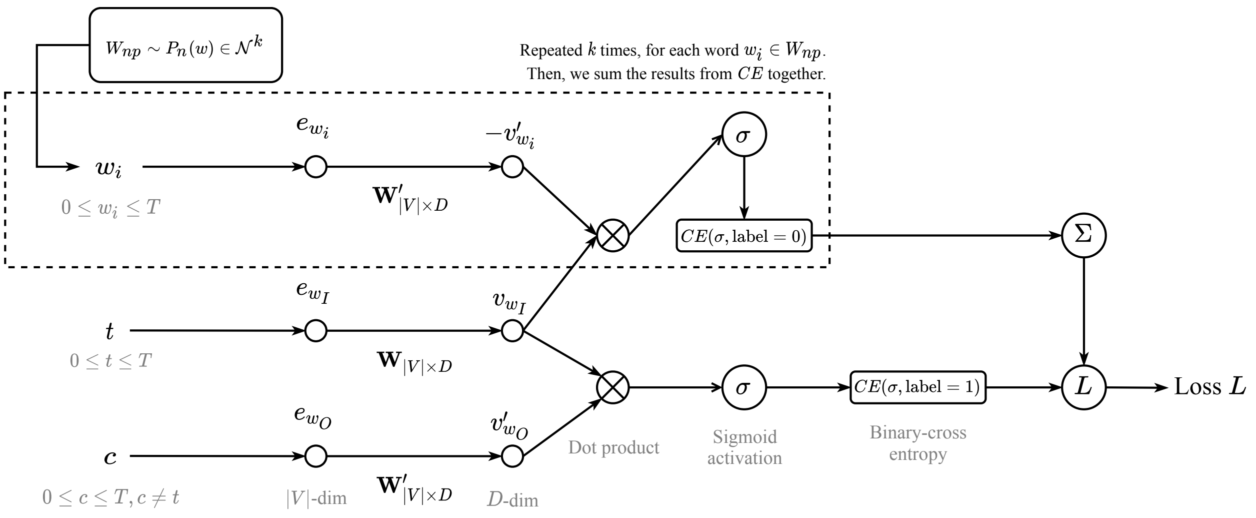
$W_{np} \sim P_n(w) \in \mathcal{N}^k$
Repeated k times, for each word $w_i \in W_{np}$. Then, we sum the results from CE together.
w_i
$0 \le w_i \le T$
e_{w_i}
$\mathbf{W}'_{|V| \times D}$
$-v'_{w_i}$
σ
$CE(\sigma, \text{label} = 0)$
Σ
t
$0 \le t \le T$
e_{w_I}
$\mathbf{W}_{|V| \times D}$
v_{w_I}
$CE(\sigma, \text{label} = 1)$
L
Loss L
c
$0 \le c \le T, c \ne t$
e_{w_O}
$\mathbf{W}'_{|V| \times D}$
v'_{w_O}
$|V|$-dim
D-dim
Dot product
Sigmoid activation
Binary-cross entropy

Figure 2.32: The artificial neural network architecture of word2vec Skip-gram model with negative sampling. The inputs to the network are t and c, i.e, index of target word w_t and context word w_c in the vocabulary. For each forward pass in the network we sample k word indices from $P_n(w)$. Furthermore, we use the k sampled word indices to compute the negative sampling loss L.

or represented using an integer corresponding to the index of the word in the vocabulary. For explanation purposes, we use the one-hot encoded representation here, but in our code implementation, we will use the latter one. We denote the input layers to be $\{e_{w_t}\}$ for the target word w_t and $\{e_{w_{t+j}}\}$ for the context word w_{t+j}. Each of the input layers has $|V|$ values, where $|V|$ is the size of the word vocabulary.

Hidden layer

We have one hidden layer for each input layer in our ANN. To calculate the result from the input layers to its hidden layer, we introduce two $D \times |V|$ weight matrices W and W', where D is the hidden embedding dimension and $|V|$ is the vocabulary size. The W matrix consists of weights related to the target word w_t and can be thought of as the "input to hidden" matrix. The W' matrix consists of weights related to the context word, and unlike the W matrix, it can be thought of as the "hidden to output" matrix from the original introduction to negative sampling. Note that we initialize both W and W' at random. Furthermore, we refer to W and W' as the target and context embedding matrix, respectively.

To map from the input to the hidden layer, we use a linear activation function (i.e. $id(x) = x$) with no bias, leading to more efficient training of bigger datasets. In other words, we simply multiply the embedding matrix (either W or W') with its respective one-hot encoded input vector, resulting in a "look-up" of the embedding vector. To illustrate with an example, imagine if we had the one-hot encoded vector $e_{w_t} = \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right)$ and the embedding matrix $W = \left(\begin{smallmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{smallmatrix}\right)^\top$. If the multiply the embedding matrix with the one-hot encoded vector, i.e. $W \cdot e_{w_t}$, we would get $\left(\begin{smallmatrix} 1 & 2 & 3 \end{smallmatrix}\right)^\top$, essentially performing a "copy" operation. We let $\{v_{w_t}\} = W \cdot \{e_{w_t}\}$ be the hidden layer for word w_t and $\left\{v'_{w_{t+j}}\right\} = W' \cdot \left\{e_{w_{t+j}}\right\}$ be the hidden layer for word w_{t+j}.

Output layer

Recall that when we are using negative sampling, we would like to ensure that words in the same context yield similar word embeddings, and that words that are not in the same context (i.e. a target word versus a word sampled from the noise distribution $P_n(w)$) to be dissimilar. From the objective of negative sampling in Equation (2.62), we see that we use the sigmoid function on the dot product between the "input" and the "output" vectors v and v' (in our setting: $\{v_{w_t}\}$ and $\left\{v'_{w_{t+j}}\right\}$). When we take the dot product, we are essentially computing an unnormalized cosine similarity measure between the vectors (we come back to the use of cosine similarity in Section 2.2.4). The core idea is to use this dot product similarity measure and convert it into the range of $[0, 1]$ by using the sigmoid function. Particularly, we want similar vectors to have 1 as output from the sigmoid function and dissimilar vectors to have 0 as the output from the sigmoid function. When we compute the loss of the ANN, we generate k samples from the noise distribution $P_n(w)$ such that we can use them for computing the loss of the network.

For each positive target-context pair we have in our data, we create a single sigmoid output plus k sigmoid outputs for each negative word we sample from the noise distribution $P_n(w)$. Thus, we could argue that we have $(k + 1)$ outputs in our network. We note that, however, our main interest is to learn the internal embedding matrix W; we do not use the ANN for predicting whether a certain word is more or less likely to be within its contextual neighbourhood, thus, discarding the output of the network.

Similar to [Mikolov et al., 2013a], we update the embedding weights W and W' using the stochastic gradient descent optimizer (see Section 2.1.5.6), adjusting all the weights of the embedding matrices during the training of the ANN to minimize the loss in Equation (2.63). Furthermore, we use a linearly decreasing learning rate, meaning that we start with some initial learning rate lr and decrease linearly to lr_{min} until we reach the end of training. The linearly decreasing learning rate is also used by [Mikolov et al., 2013a].

2.2.2.6 Hyperparameters in word2vec

When training a word2vec model, we have to choose several hyperparameters. In this sub-subsection, we go over all choices of hyperparameters and explain them. We will explain the specific choices of hyperparameters that we use to train a word2vec model in Section 3.1.3.

- **min-word-count**
  The minimum word count denotes a threshold of how many times a word at least has to occur in a text for it to be in the vocabulary. In the empirical experiments of Mikolov et al, they used 5 as the threshold.
- **max-vocab-size**
  The maximum vocabulary size denotes the maximal number of words to have in our vocabulary; we use the top **max-vocab-size** most frequent (i.e common) words. We may set the maximum vocabulary size to reduce the computational complexity and to remove some less occurring words.
- **batch-size**
  Batch size is the number of positive target-context pairs (w_t, w_{t+j}) we train on in each training step, i.e. the number of forward passes we perform in our ANN before we do a backwards pass.
- **num-epochs**
  The number of epochs denotes the number of times we train on the training data. With word2vec, we typically set this number rather low (e.g. $1-5$), as [Mikolov et al., 2013a] reports that by training on more data, we require fewer epochs to get comparable or better quality word embeddings.
- **learning-rate**
  The learning rate denotes how fast we want our weights to change in our ANN. The original authors of word2vec used 0.025 (i.e. 2.5%) as the initial learning rate for their experiments.
- **min-learning-rate**
  The minimal learning rate denotes how small the learning rate should be when approaching the end of the training. Mikolov et al. stated that they decreased it linearly,

such that it approaches zero at the end of the last training epoch. We note, however, that in the original code of word2vec, they linearly decrease the learning rate to the initial learning rate lr times 0.0001 (i.e. $0.025 \times 0.0001 = 0.0000025$) [Mikolov, 2013c, line 398].

- **embedding-dim**
  The embedding dimension denotes the dimension we want to use for the internal matrices W and W' in our ANN, i.e. the dimensionality of the word embeddings.
- **max-window-size**
  Maximum window size denotes the maximal number of words to look for to the left and the right of a target word w_t. Mikolov et al. reported that they used 5 as the window size.
- **num-negative-samples**
  The number of negative samples denotes how many negative samples we generate for each positive target-context pair we train on.
- **sampling-factor**
  We use the sampling factor as a threshold to randomly discard frequently occurring words in the text corpora. A common value for this is 10^{-5}.
- **unigram-exponent**
  The unigram exponent is which power we raise the noise distribution $P_n(w)$ to (where the noise distribution equals the unigram distribution, in our case). Although there was no theoretical justification for this, Mikolov et al. reported that the value 3/4 worked the best.

2.2.3 Other models for learning word embeddings

Creating word embeddings is a task that can be achieved in various ways. In this subsection, we briefly introduce two different models for computing word embeddings. The first model is the Global Vectors (GloVe) [Pennington et al., 2014] model. GloVe learns vector representations for words in a more ”explicit” fashion than word2vec for example; we come back to this in Section 2.2.3.1. The second model is the fastText [Bojanowski et al., 2017] model, which is an extension of the original word2vec Skip-gram model to include sub-word information. Note that we primarily focus on word2vec using Skip-gram with negative sampling in this thesis, and for this reason, we will not go into equal depth when explaining

GloVe and fastText. We will use pre-trained GloVe and fastText models when comparing evaluation results of word embedding models in Section 3.1.4 and when we analyze word embeddings in Section 3.3.1.

2.2.3.1 GloVe

Global Vectors (GloVe) [Pennington et al., 2014] is a model for learning vector representations for words. In contrast to word2vec, GloVe trains on the word to word co-occurrence counts, and thus, makes efficient use of statistics. In addition to this, the objective of GloVe is more explicit, as opposed to the vector representations word2vec learns, which are merely a by-product of the training. We base this sub-subsection on [Pennington et al., 2014].

To understand how GloVe works, we first introduce the notion of the *word to word co-occurrence matrix*, which we denote X. X is a square matrix where each element X_{ij} represent the number of times word j occurs in the context of word i. Following, let $|V|$ denote the number of unique words in the vocabulary and let $X_i = \sum_{k=1}^{|V|} X_{ik}$ be the number of times any word appears in the context of word i. Using X_{ij} and X_i, we can establish a probabilistic model P_{ij} of how often a given word j falls in the context of word i. Finally, we let $P_{ij} = P(j|i) = X_{ij}/X_i$. To motivate the use of P_{ij}, imagine that we want to investigate the concept of temperatures, which we extract directly from the co-occurrence probabilities. Consider the words $i =$ sunny and $j =$ cloudy. We can explore relationships of the words i and j by studying the ratios of the co-occurrence probabilities with various other words, k. If we set the word $k =$ hot, we expect the ratio P_{ik}/P_{jk} to be large, since intuitively, the word "heat" is more related to the word "sunny" than the word "cloudy". If the word k is set to an unrelated word of both sunny and cloudy, the ratio should be around 1, as both probabilities become rather low. The authors of [Pennington et al., 2014] give an example of studying the ratios of co-occurrence probabilities and is the foundation of how GloVe incorporates word count statistics to learn vector representations.

To learn vector representations of words, Glove uses two weight matrices, which we denote $W = \left\{w_1, w_2, \ldots, w_{|V|}\right\} \in \mathbb{R}^{|V| \times d}$ and $\tilde{W} = \left\{\tilde{w}_1, \tilde{w}_2, \ldots, \tilde{w}_{|V|}\right\} \in \mathbb{R}^{|V| \times d}$, similar to word2vec. That is, the weight matrix W represents the target word embeddings, while $\tilde{W}$ represent the context word embeddings. Furthermore, the objective function of GloVe consists of a weighted squared loss J and we define it as

$$J = \sum_{i,j=1}^{|V|} f\left(X_{ij}\right) \left({w_i}^\top \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2 , \tag{2.67}$$

where f is a weighting function, b_i is bias for w_i and $\tilde{b}_j$ is bias for $\tilde{w}_j$. The authors of GloVe found a particular class of functions for f that was suitable, and we define it as

$$f(x) = \begin{cases} (x/x_{\max})^\alpha & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases} , \tag{2.68}$$

where α and $x_{\max}$ are hyperparameters. In the experiments performed in [Pennington et al., 2014], the authors let $x_{\max} = 100$ and $\alpha = 3/4$. Furthermore, to train the GloVe model, they iteratively learn the weights over time in a gradient descent fashion, using AdaGrad [Duchi et al., 2011] in particular, with an initial learning rate of 0.05. Finally, GloVe uses the sum of its weight matrices, i.e. $W + \tilde{W}$, as the word embeddings, which the authors claim to lead to a minor increase in performance.

2.2.3.2 fastText

fastText is an extension the word2vec Skip-gram model with negative sampling [Bojanowski et al., 2017]. In particular, fastText represent each word using character n-grams, i.e. sub-words of length n (e.g. *que* is a 3-gram of the word *quest*). We associate vector representations to each character n-gram, and following, we vectorize words using the sum of such representations. By creating vector representations of character n-grams, fastText can create representations for words that are not in the training vocabulary. We base this sub-subsection on [Bojanowski et al., 2017].

Recall the objective function of the Skip-gram model with negative sampling, which we show in Equation (2.63). The authors of fastText generalize it by replacing the dot product between word embeddings with a scoring function $s(w_I, w_O) \mapsto \mathbb{R}$. We generalize Equation (2.63) using scoring function s and define it as

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{w_O \in cw(w_I)} \log\sigma\left(s(w_I, w_O)\right) + \sum_{w_i \in W_{np}} \log\sigma\left(-s(w_I, w_i)\right), \tag{2.69}$$

where $s(w_I, w_O) = {v'_{w_O}}^\top v_{w_I}$ for the Skip-gram model with negative sampling.

As we mention in the introduction of this sub-subsection (see Section 2.2.3.2), fastText vectorizes each word w using the vector representation of its character n-grams. To indicate the start and end of a word, we use the characters $<$ and $>$, respectively. We do this such that that the model can distinguish between prefixes and suffixes of words. Now, to give an example, let $w =$ carbon and $n = 3$. We represent the word w by character n-grams:

$$<\text{ca}, \text{car}, \text{arb}, \text{rbo}, \text{bon}, \text{on}>,$$

in addition to the word itself <carbon>. Note that the n-gram <car>, from the word car, is different from the n-gram car of the word w, due to the prefix and suffix characters. Furthermore, let $\mathcal{G}_w \subset \{g_1, g_2, \ldots, g_G\}$ be the set of n-grams appearing in word w, where G is the total number of n-grams of all words in the vocabulary. We associate a vector z_g to each entry g in $\mathcal{G}_w$. The goal of fastText is to represent each word embedding as a sum of its n-grams, and consequently, we obtain the scoring function s. We define the scoring function s as

$$s(w_I, w_O) = \sum_{z_g \in \mathcal{G}_{w_I}} {z_g}^\top v_{w_O}. \tag{2.70}$$

This minor change to the scoring function allows the fastText model to share vector representations of n-grams between words, and thus, allow to learn more accurate representations for rare words (i.e. words that occur rarely in the training vocabulary).

Similar to the training of word2vec, the authors of fastText use stochastic gradient descent on the objective function in Equation (2.69) with Equation (2.70) as scoring function. For more details of the training process, we kindly refer the reader to [Bojanowski et al., 2017, p. 3-4].

2.2.4 Evaluating word embedding models

In typical machine learning projects, we commonly set aside some portion of the data, e.g. the test data set, and then use this data to evaluate the unbiased performance of models. In word embedding models, however, we do not follow this kind of practice. Instead, we evaluate the quality of the word embeddings on test data sets that measure word relatedness, as our interest lies in how well the word embeddings relate to each other. An example of a word relatedness test could be to check whether or not the word *Oslo* is related to *Norway* as the word *Rome* is related to *Italy*. In practice, however, for such word relatedness tests, we do not know the word *Italy* (i.e. we hide it), and we must guess it from the entire vocabulary. By accumulating several word relatedness tests, we can create test sets, which we refer to as *analogy data sets*. More generally, analogy data sets consists of questions to check whether or not a word A is related to B as C is related to D. By vectorizing the words, we want the differences between each pair of vectorized words to be roughly equal, and we show this relation as

$$v_B - v_A \approx v_D - v_C. \tag{2.71}$$

Solving Equation (2.71) for v_D, we get that

$$v_D \approx v_B - v_A + v_C. \tag{2.72}$$

Now, since v_D is "hidden" from the model, we have to find a way of searching for the closest word matching the right-hand-side of Equation (2.72). To do so, we often use cosine similarity, both for analogy tasks and measuring the distance between any two word embeddings. Let u and v be two vectors of the same size. We define the dot product between the vectors u and v as

$$u \cdot v = ||u|| \, ||v|| \cos(\theta), \tag{2.73}$$

where $|| \cdot ||$ is the magnitude (length) of the vector and $\cos(\theta)$ is the cosine of the angle θ between u and v, which we refer to as the *cosine similarity*. By solving for $\cos(\theta)$ in Equation (2.73), we get

$$\cos(\theta) = \frac{u \cdot v}{||u|| \, ||v||}. \tag{2.74}$$

Thus, we see that by computing the cosine similarity of two vectors u and v, we discard their magnitude, and thus, both vectors both have unit length. The removal of the vector magnitude is important, as we would like to compare vectors using vector addition and subtraction, as motivated by Equation (2.71). Using cosine similarity, we can find the closest matching word embedding v_D (and associated word D) by computing

$$v_D = \underset{v_{\tilde{D}} \in W^*}{\arg\max} \left(\cos\left(v_{\tilde{D}}, v_B - v_A + v_C\right)\right), \tag{2.75}$$

where W^* are the word embeddings W in which we exclude the word embeddings v_A, v_B and v_C. It is common to exclude the vectors v_A, v_B and v_C from the search, as [Mikolov et al., 2013a] did in their experiments.

Many machine learning algorithms require us to use Euclidean distance for comparing distances between vectors. To convert from cosine similarity to Euclidean distance, we observe that there is a relationship between the two vectors u and v. We show this relationship as

$$||u - v||_2^2 = (u - v)^\top (u - v) = ||u||^2 + ||v||^2 + 2u^\top v, \tag{2.76}$$

where $|| \cdot ||_2^2$ is the squared Euclidean distance. Let $||u|| = ||v|| = 1$, i.e the vectors u and v are of unit length, then Equation (2.76) becomes

$$\begin{aligned} ||u - v||_2^2 &= (u - v)^\top (u - v) \\ &= 2 + 2u^\top v \\ &= 2(1 + u^\top v) \\ &= 2(1 + \cos(u, v)). \end{aligned} \tag{2.77}$$

In other words, we see a clear relationship between the squared Euclidean distance and cosine similarity. Some machine learning algorithms are applicable using dot-product as the distance metric, and by an expansion of Equation (2.74), we see a particular relationship for unit-length vectors u and v, namely that

$$\cos(\theta) = u \cdot v. \tag{2.78}$$

Furthermore, by using normalized word embeddings, we may use (squared) Euclidean distance or dot-product to emulate cosine similarity, as we get the same distance ranking by using cosine similarity. Finally, we define the *cosine distance* to be $(1 - \cos(\theta))$. We use the cosine distance in algorithms for measuring the distances between word embeddings. Furthermore, we will use cosine similarity when evaluating word embeddings models in Section 3.1.4.

2.3 Topological data analysis

Topological data analysis (TDA) is a fast-growing field of mathematics, providing a set of tools from topology to infer underlying features from data [Chazal and Michel, 2021]. In this section, we will introduce relevant concepts from TDA. In particular, we will introduce the simplicial complex in Section 2.3.1 and persistence diagrams in Section 2.3.2. Furthermore, we will introduce a method for vectorizing persistence diagrams, namely the persistence image in Section 2.3.3, and a commonly used distance metric for persistence diagrams, namely the p-Wasserstein distance in Section 2.3.4. Finally, we will introduce two algorithms that uses concepts from TDA to identify singular words (or data points), namely topological polysemy in Section 2.3.5 and Geometric Anomaly Detection in Section 2.3.6. This section is based on [Edelsbrunner and Harer, 2010,Chazal and Michel, 2021], if not stated otherwise.

2.3.1 Simplicial complex

In computer science, a *graph* is a datatype for describing possibly non-linear and complex relationships between data points. In particular, graphs consist of vertices and edges, where the edges connect the vertices. Edges can also have metadata such as weight and direction. A common graph to use (in the context of computer science) is the k-nearest neighbour graph, where vertices are data points and edges represent neighbouring relationships, with distance as weight. A *simplicial complex* is a generalization of graphs and we see its particular usage in TDA, due to its topological properties. Simplicial complexes consist of *n-simplices*, where 0-simplices are similar to vertices, 1-simplices are similar to edges. The difference between graphs and simplicial complexes occur when we look at n-simplices for $n \geq 2$. For example, 2-simplices form triangles and 3-simplices form tetrahedrons (i.e. triangular pyramids). We show examples of n-simplices in Figure 2.33.

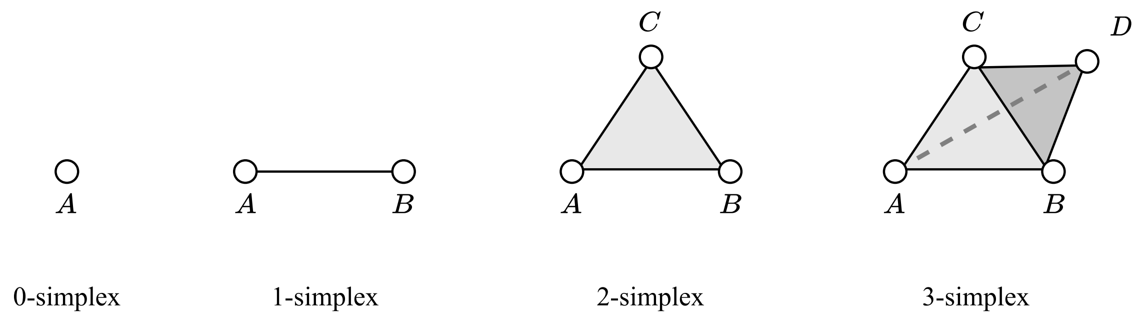

Figure 2.33: Building blocks of simplicial complexes, consisting of n-simplices.

By combining one or more n-simplices, we form simplicial complexes. We illustrate an example of a simplicial complex in Figure 2.34.

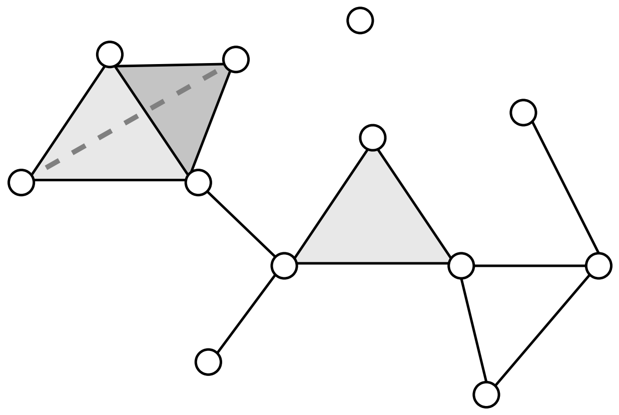

Figure 2.34: Simplicial complex consisting of 0-, 1-, 2- and 3-simplices.

There exist several methods for creating simplicial complexes from data. In particular, we look at one such simplicial complex in the next sub-subsection, namely the Vietoris–Rips complex.

2.3.1.1 Vietoris–Rips complex

The *Vietoris–Rips complex* is a simplicial complex that we create from data points using any distance metric. Let α be a *proximity diameter*. We build the Vietoris–Rips complex by forming a simplex for every set of k data points with distance less than or equal to α. That is, if k data points satisfy $d(x_i, x_j) \leq \alpha$ (where $d(x_i, x_j)$ computes the distance between two data points i and j), we create a $(k-1)$ simplex for that particular data point (1-simplex for two data points, 2-simplex for three data points, etc.). We illustrate with an example of a Vietoris–Rips complex in Figure 2.35.

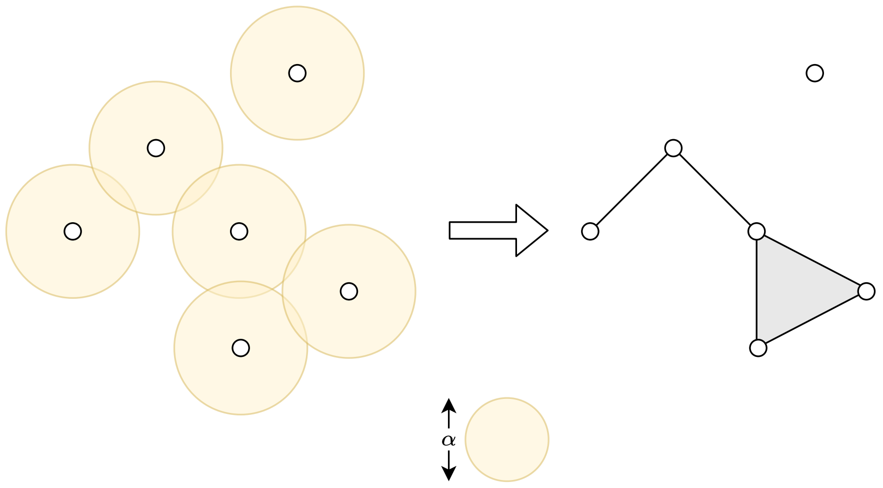

Figure 2.35: A Vietoris–Rips complex on 2-dimensional data with proximity diameter α.

By carefully studying the n-simplices of a Vietoris–Rips complex, we can observe topological structures such as *loops* (i.e. 2-simplices) or *holes* (i.e. 3-simplices). We show an example of a loop in Figure 2.35, where the rightmost data points at the bottom form a 2-simplex. To study the topological properties of data, it is common to look at varying values of α, starting at zero and increasing to infinity. In particular, we use what we call *persistent homology*, which is the study of observing how the topology changes once a threshold (e.g. α) increases. To easily present and visualize persistent homology, we use persistence diagrams. We look at persistence diagrams in the next subsection.

2.3.2 Persistence diagram

To present persistent homology of a simplicial complex, we use *persistence diagrams*. In persistence diagrams, we look at a range of proximity diameters over multiple persistent *homology dimensions* (or *homology degrees*), where the dimension refer to which n-simplices we want to look at (0-dimensional persistent homology observe changes of 0-simplices, 1-dimensional observe changes of 1-simplices, etc.). A persistence diagram is 2-dimensional, where the x-axis denotes the birth time and the y-axis the death time. Assuming that we want to create a persistence diagram of data points, we let the proximity diameter α start at zero. By gradually increasing α, new topological properties appear and old topological properties merge into other properties. In particular, once a topological property is "birthed" (e.g. 1-simplex between two data points), the birth time is noted for the particular data point. A point in the persistence diagram appears once n-simplices merge into new m-simplices (e.g. three 1-simplex becoming a 2-simplex).

To motivate the use of persistence diagrams, we illustrate with a simple example in Figure 2.36, where we study the change of 0-dimensional persistent homology. In Figure 2.36, we see how the persistence diagram changes once we increase the proximity diameter α, on a data set consisting of two blobs. In Figure 2.36 (a), we let $\alpha = 0.2$ and we observe that there are only two data points intersecting, thus leading to a single point in the persistence diagram. In Figure 2.36 (b), we let $\alpha = 1.5$ and we observe how the data points in each blob connect, thus leading to several entries in the persistence diagram. In Figure 2.36 (c), we let $\alpha = 4.5$, and we observe that there is a single point and several points on the bottom in the persistence diagram. By looking at the plot on the left, we see how the two blobs intersect and all data points have connections to other data points with distance less or equal to α. In addition to this, Figure 2.36 (c) indicates that we have two clusters in our data, although our data is rather noisy and could be more compact.

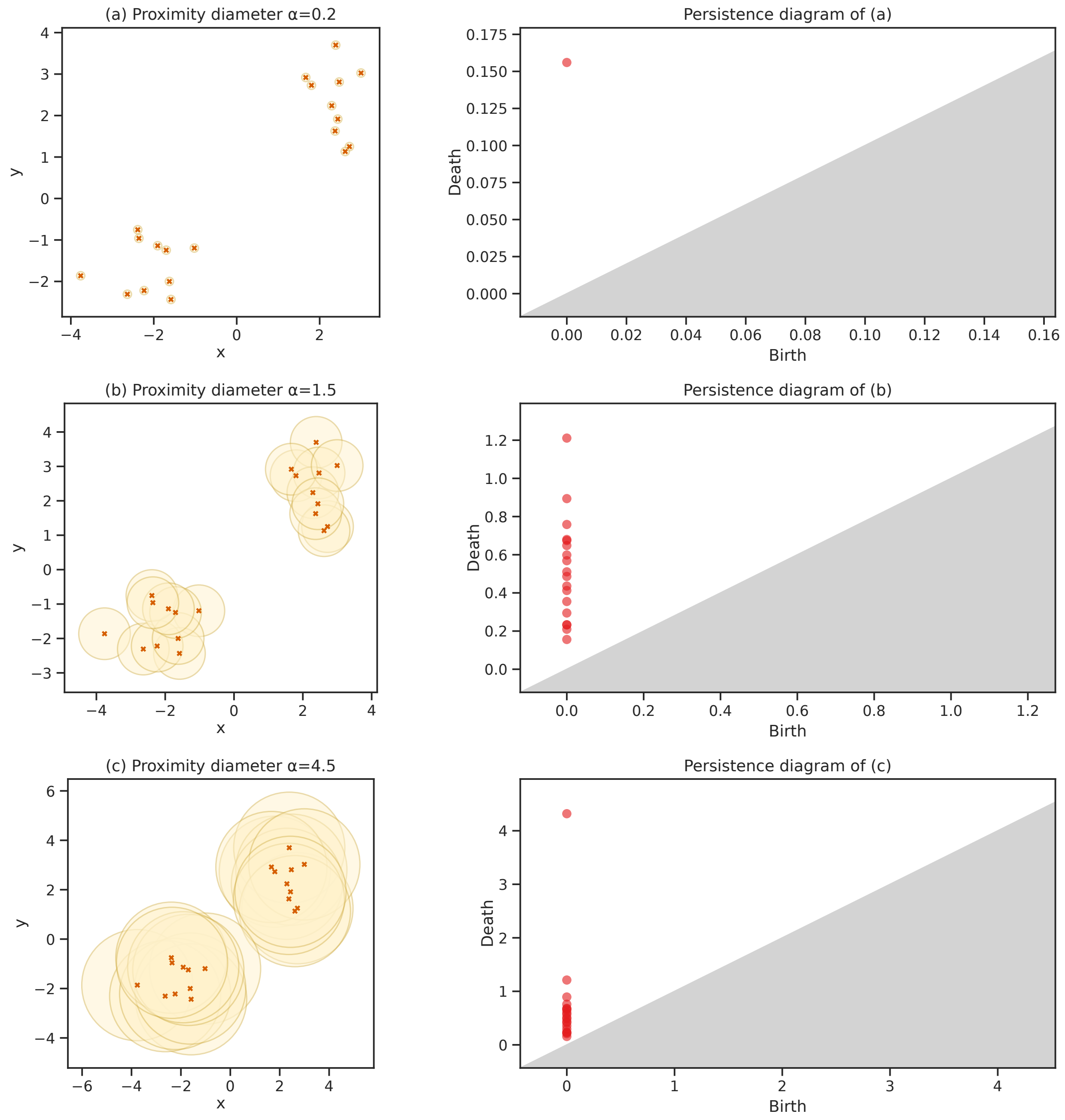

Figure 2.36: Persistence diagrams computed of a data set of two blobs. We vary the levels of α, showing how the topological properties form in the persistence diagrams.

An important aspect of machine learning is to predict some quantity given some features. If we want to use the persistence diagrams as features in a model, for instance, a typical approach would be to perform some feature extraction or vectorization first. An example of vectorization of persistence diagrams is persistence images, which we introduce in the next subsection.

2.3.3 Persistence image

Many machine learning tasks require valuable features to yield good results. *Persistence images* [Adams et al., 2016] are vector representations of persistence diagrams. When explaining persistence images, we refer to [Adams et al., 2016]. Furthermore, we explain the use of persistence images for the analysis of word embeddings in Section 3.3.2 and when discussing future work in Chapter 5.

Let $B = \{(b_1, d_1), (b_2, d_2), \dots, (b_m, d_m)\} \in \mathbb{R}^{m \times 2}$ be a persistence diagram in birth-death coordinates. First, we transform B into birth-persistence coordinates, where *persistence* is the difference between death and birth. That is, let $T(B) = \{(b_1, p_1), (b_2, p_2), \dots, (b_m, p_m)\} \in \mathbb{R}^{m \times 2}$, where $p_i = b_i - d_i$ for $1 \leq i \leq m$. Following, for each of the point in the the $T(B)$ persistence diagram, we place a probability distribution. A common choice is to use the Gaussian distribution, which we center at each data point respectively. By placing probability distributions on each point, we are able to differentiate between dense and sparse areas. In addition to placing probability distributions on each point, we weight the distributions by the persistence of the point, making more persistent areas more prominent than others. Using the weighted probability distributions, for a particular persistence diagram B we define the persistence surface of ρ_B as

$$\rho_B(z) = \sum_{u \in T(B)} f(u)\phi_u(z), \tag{2.79}$$

where $f(u)$ is the persistence weighing function and $\phi_u(z)$ is the probability distribution which we evaluate at point z (e.g. if Gaussian, then we centre it at point u). Finally, we reduce the persistence surface $\rho_B(z)$ to a discretized representation. In particular, we form an $N \times M$ grid, and for each cell in the grid, we compute the integral of $\rho_B(z)$ over that region and use the result from the integral as value. In other words, this discretization allows us to summarize the persistence surface using less information and we are left with an $N \times M$ matrix which we can for machine learning tasks more easily. Following, we illustrate the use of persistence images in Figure 2.37, where we apply it to a 2-dimensional data set consisting of two circles. In Figure 2.37, we see how the data is transformed into a persistence diagram B, following by the transformed persistence diagram $T(B)$ and finally the persistence image of B. In Figure 2.37 (d), we see a 66 by 35 pixels image, representing the vectorization of persistence diagram B.

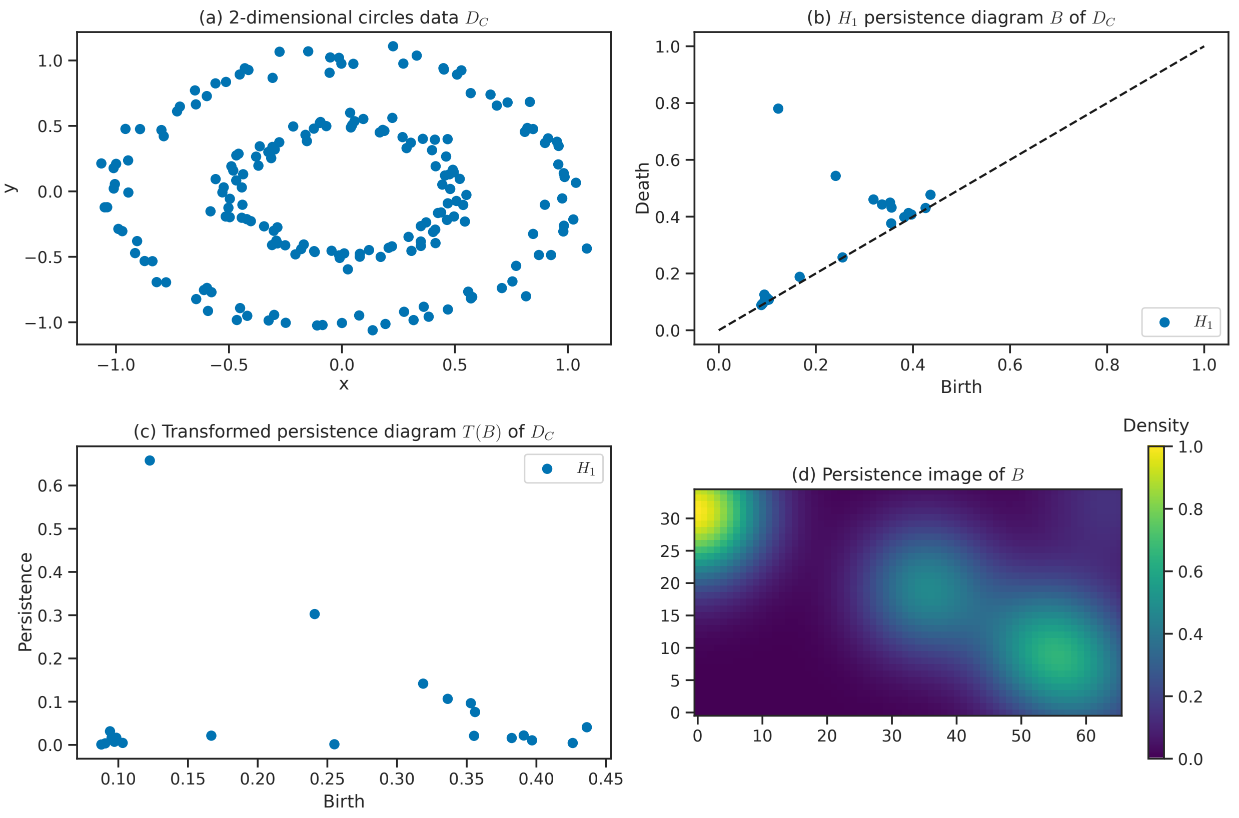

Figure 2.37: Persistence image pipeline when applied to a 2-dimensional circles data set.

2.3.4 Wasserstein distance

When we compare distances between data points, we commonly use distance metrics (e.g. Euclidean distance). To compare differences between any two persistence diagrams A and B, however, we use the p-*Wasserstein distance*, which we define as

$$W_p(A,B) = \inf_{\gamma:A\to B}\left(\sum_{u\in A}||u-\gamma(u)||_\infty^p\right)^{1/p}, \tag{2.80}$$

where $1 \leq p < \infty$ and γ ranges over bijections between A and B, and inf is the infimum (i.e. greatest lower bound; similar to "minimum"). We further visualize the idea behind the p-Wasserstein distance in Figure 2.38, where we see how the points in each persistence diagram A and B get paired up. We match points to the diagonal if we find no matches for the particular point.

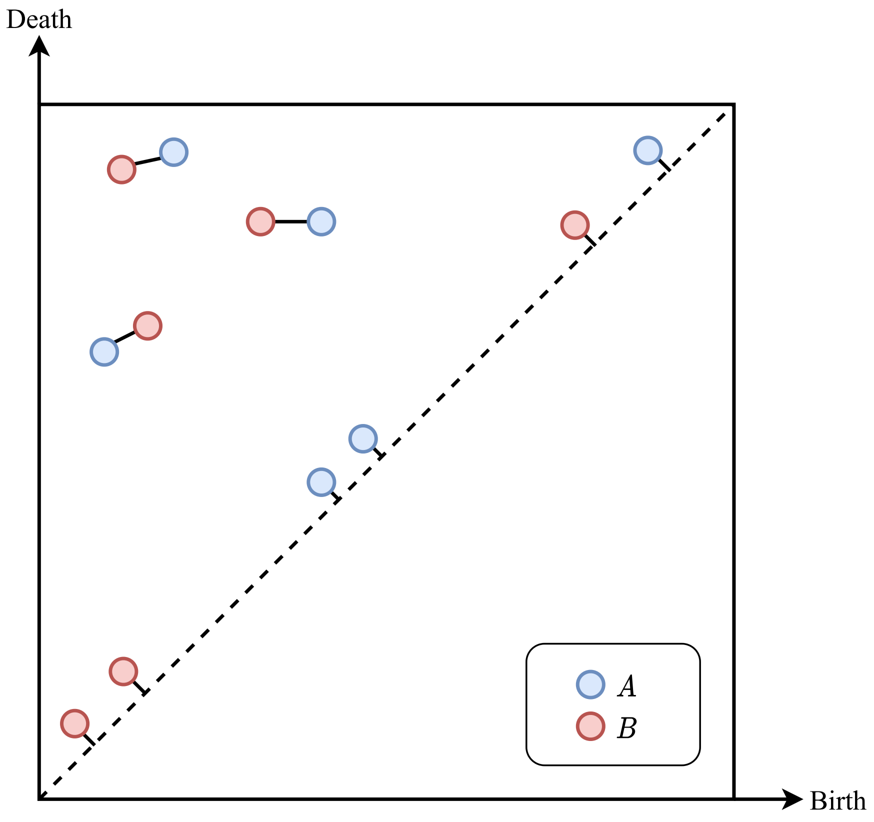

Figure 2.38: p-Wasserstein distance between two persistence diagrams A and B.

Finally, if we let $p = \infty$, i.e $W_\infty(A, B)$, we get what we call the *bottleneck distance*. We use the bottleneck distance to compare persistence diagrams as well. However, we note that the bottleneck distance has the disadvantage of only using the maximum distance between any two points over the bijections between A and B.

2.3.5 Topological polysemy

Recall the manifold hypothesis, which states that, in general, real-life high-dimensional data tends to live on a low-dimensional submanifold embedded within the high-dimensional space

[Bengio et al., 2014, p. 16]. To give an example, imagine that we some data about weight and height of humans. Naturally, we see that as the height increases, the weight increases as well. That is, we have a strong linear relationship (or correlation) between height and weight. We can therefore argue that the manifold dimension of the data is 1, even though the original data has dimension 2. The hypothesis should, in theory, also apply to word embeddings, but the authors of [Jakubowski et al., 2020] argue that word embeddings, should instead, live on a *punched manifold*. By a pinched manifold, we mean a manifold where we "glue" together particular points that are equal in some sense, creating *singular* areas in the manifold. We show an example in Figure 2.39, where we see an ideal pinched manifold for the word "solution" and four of its meanings as *submanifolds*. For word embeddings, [Jakubowski et al., 2020] claim that these singular areas in the manifold represent polysemous words and that the neighbours of polysemous words share some relation to the polysemous word. To identify such polysemous words from word embeddings, [Jakubowski et al., 2020] introduce a topological measure of polysemy (using concepts from persistent homology) that correlates well with the true number of meanings of a word. To explain the topological measure of polysemy, we refer to [Jakubowski et al., 2020].

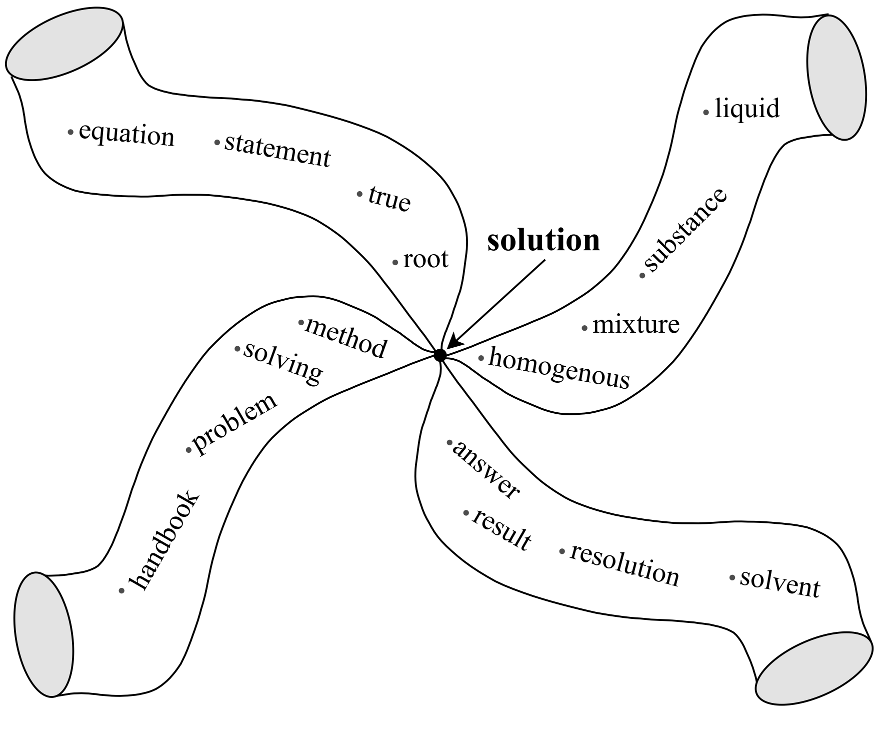

Figure 2.39: A pinched manifold of the word "solution", showing four of its meanings as submanifolds. This figure is inspired by [Jakubowski et al., 2020, Figure 5].

Determining the number of word meanings is a non-trivial task. Consider the word *solution*, which has multiple meanings. In some contexts, the word *solution* can relate to problem-solving (e.g. solving a problem in a handbook), while in other contexts, it can relate to chemistry (e.g. a mixture of two or more substances). We call such words *polysemous*, meaning that the word has multiple meanings which can relate to one another. The opposite of a polysemous word is a *monosemous* word, meaning that the word only has one meaning. The motivation behind the topological measure of polysemy, as introduced by [Jakubowski et al., 2020], stamps from the fact that the number of components of a *punctured neighbourhood* around a word w should reflect the number of word meanings of the word w. A punctured neighbourhood of w is the neighbouring words of w excluding the word w itself.

Let $W \in \mathbb{R}^{|V| \times d}$ be word embeddings, where $|V|$ is the number of words in the vocabulary and d is the word embedding dimension. We compute the *topological polysemy* $\text{TPS}_n(w)$ by fixing a target word w and a neighbourhood size n. We denote the word embedding of w as $v_w \in \mathbb{R}^d$. To compute $\text{TPS}_n(w)$, we first normalize the word embeddings W such that they are of unit length. We denote the normalized word embeddings as W_{norm} and the normalized word embedding of the target word w as $v_{w_{\text{norm}}}$. Following, we compute the punctured neighbourhood $N_n(w)$, that is, the neighbouring n normalized word embeddings around $v_{w_{\text{norm}}}$, excluding w itself. Furthermore, we project the word embeddings of $N_n(w)$ to lie at the unit sphere, with $v_{w_{\text{norm}}}$ as the center. We denote this normalized punctured neighbourhood as $N'_n(w)$. In other words, we make the word embedding of w to be the origin of a d-dimensional sphere and project the neighbouring words of w to lie around it. Finally, we compute the 0-degree persistence diagram of $N'_n(w)$ and denote it as $PD_n(w)$. $\text{TPS}_n(w)$ is then the 1-Wasserstein distance (see Section 2.3.4) between $PD_n(w)$ and the empty persistence diagram, also known as the Wasserstein norm. Furthermore, we will use topological polysemy later when analysing word embeddings in Section 3.3.1 and for prediction of polysemous words in Section 3.3.4.

2.3.6 Geometric Anomaly Detection

The manifold hypothesis forms a foundation of modern data science. Many manifold learning and dimensionality reduction algorithms rely on this assumption to find meaningful low-dimensional representations of high-dimensional data. Examples of such algorithms include PCA and UMAP (see Section 2.1.4.1 and Section 2.1.4.2). *Geometric Anomaly Detection*

(GAD) [Stolz et al., 2020] is an algorithm for identifying possible points in data that fail to satisfy the manifold hypothesis. Following, we describe the motivation behind the GAD algorithm and describe how it works. We refer to [Stolz et al., 2020] when explaining the GAD algorithm.

We first introduce the motivation behind the GAD algorithm to deepen our understanding of how it works. Imagine that we have some data that lies on two submanifolds P and Q. We illustrate such a situation in Figure 2.40. Here we assume that the two submanifolds P and Q are planes that intersect, which we mark by the dotted red line. We place an *annulus* around each data point and infer its topological structure. An annulus is a region between two circles, where the first circle is contained in the other, and both circles share a centre point. Annuli can also remind us of rings. Formally, each annulus around its respective data point has an inner radius r and outer radius s. Depending on where points are on either of the submanifolds, we observe that they can have one of three states as seen in Figure 2.40 (a), (b) and (c). If a point is at the boundary of either submanifold, i.e. in Figure 2.40 (a), we observe that the points falling into the annulus around the data point forms a half-circle, as half of the circle does not have any data points in them. If we count the number of topological loops we get zero. If the data point falls nicely into either submanifold, i.e. in Figure 2.40 (b), we observe that we get a nice annulus where neighbouring points falling into the annulus around the data point forms a circle. In other words, if a point is on the submanifold, we expect to get exactly one topological loop. In the last situation, i.e. Figure 2.40 (c), we have a data point that falls between P and Q, creating a singularity (or anomaly) in the data. This stamps from the fact that it is harder to distinguish which submanifold the particular data point should belong to. If we look at the data points in the annulus around the data point in Figure 2.40 (c), we observe that we get two or more topological loops.

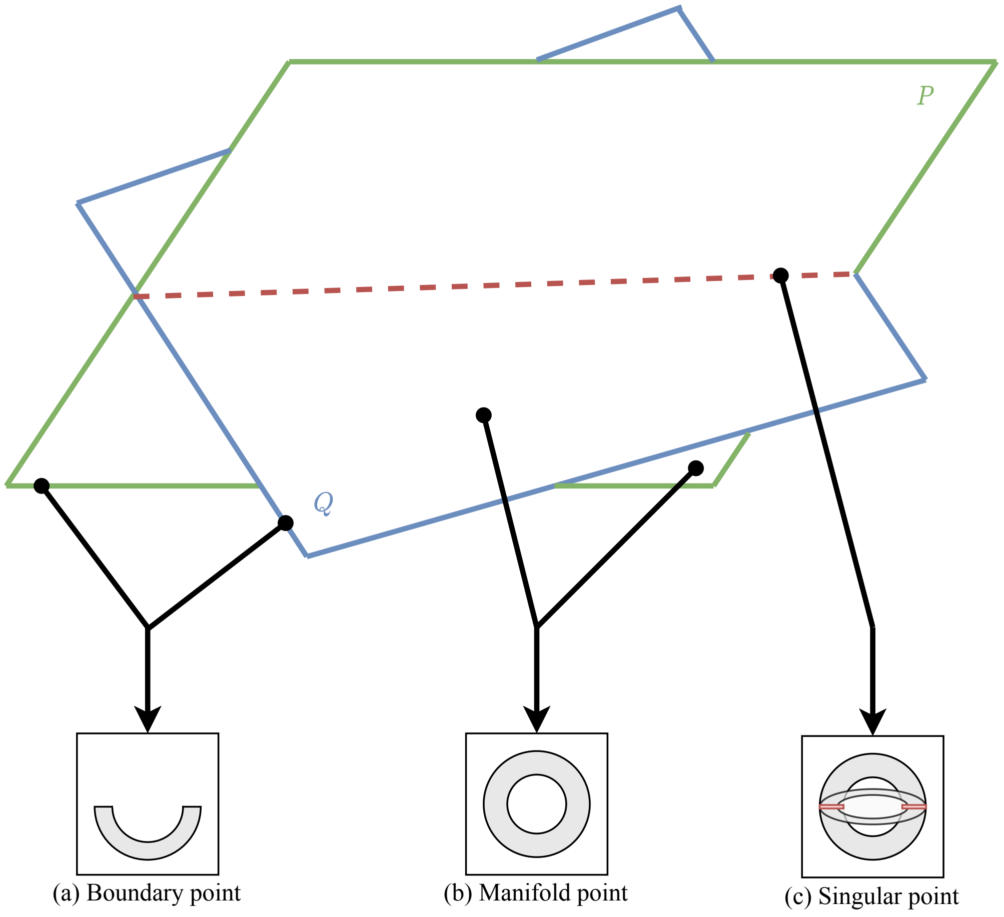

Figure 2.40: The motivation behind the GAD algorithm, illustrated. Data points belonging to two submanifolds P and Q, and depending on where the data points are on the submanifolds, it can have three different states: (a), (b) or (c). The GAD algorithm is particularly interested in finding singular (c) data points between k submanifolds (here: $k = 2$). This figure is inspired by [Stolz et al., 2020, Figure 1].

Let $X \in R^{n \times d}$ be data points. The GAD algorithm works as follows: fix two parameters $0 < r < s$, and for each data point $x_i \in X$ place an annulus around it, with inner radius r and outer radius s. We determine the data points that fall into the *annular neighbourhood* of x_i and denote this set as $A_y = \{a_1, a_2, \ldots, a_m\} \subset X$. That is, the set A_y consists of those points a_j that satisfies $r \leq d(x_i, g_j) \leq s$, where $d(\cdot, \cdot)$ measures the distance between two points (e.g. using Euclidean distance). Then, select a manifold dimension k we would like to investigate, as the GAD algorithm discovers intersections of $(k-1)$ submanifolds. To find intersections of $(k-1)$ submanifolds of the annular neighbourhood A_y of x_i, we compute the $(k-1)$-dimensional Vietoris–Rips complex of A_y, which we denote $VR_{k-1}(A_y)$.

Then, for each birth-death coordinate $(b_{k-1}, d_{k-1}) \in VR_{k-1}(A_y)$, we count the number of points that persist longer than the annulus width. We denote this count as N_y. Recall that the persistence of points in persistence diagrams can be computed by transforming the persistence diagram into birth-persistence coordinates, where persistence $p_{k-1} = d_{k-1} - b_{k-1}$. We define the annulus width to be the difference between the outer and inner radius, i.e. $w_{A_y} = s - r$. To count N_y, we iterate over the points in the persistence diagram and count the number of points that satisfies

$$p_{k-1} > w_{A_y}. \tag{2.81}$$

The count N_y is analogous to the number of topological loops that occur in A_y, for a particular homology dimension $(k-1)$. If no points satisfy Equation (2.81), i.e. $N_y = 0$, then we classify x_i as a boundary point, similar to the situation in Figure 2.40 (a). If we have exactly one point that satisfies Equation (2.81), i.e $N_y = 1$, then we classify the point as a boundary point, similar to the situation in Figure 2.40 (b). If two or more points satisfy Equation (2.81), i.e. $N_y > 1$, then we classify the point as a singular point, similar to the situation in Figure 2.40 (c). Furthermore, we will use GAD for later analysis of word embeddings in Section 3.3.2 and for prediction of polysemous words in Section 3.3.4.

Chapter 3

Analysis of Word Embeddings

In this chapter, we will use methods from machine learning to analyze word embeddings. Due to the scope of the thesis, we will mainly analyze word embeddings from the word2vec model (Section 2.2.2) using Skip-gram and negative sampling. We will also run some of the analysis methods on published word embeddings from external papers, in particular in Section 3.3.

Firstly, we will describe how we trained and evaluated our word2vec implementation. In particular, we will explain the data preprocessing steps, the implementation specifics and hyperparameter choices. We will also show how we evaluated our trained word2vec model. Secondly, we will perform cluster analysis on word embeddings to look for deeper structure. In particular, we will compare clustering algorithms trained on word embeddings, using internal cluster validation methods, and investigate the clustering of distinct groups of words. Thirdly, we will look at the application of two methods from topological data analysis (TDA) on word embeddings. Lastly, we end the chapter by creating two supervised models for estimating the number of word meanings, using the results from TDA and intrinsic dimension estimation. We train the supervised models and visualize their evaluation results.

To perform the analyses in this chapter, we utilized the Python programming language with some key Python packages: `numpy` [Harris et al., 2020] (efficient vector and matrix manipulation), `scikit-learn` [Pedregosa et al., 2011] and `scipy` [Virtanen et al., 2020] (general methods from machine learning), `matplotlib` [Hunter, 2007] and `seaborn` [Waskom, 2021] (tools for data visualization), `joblib` [Joblib Development Team, 2021] (data dumping to file), `sharedmem` [Feng et al., 2020] (parallelization of trivial jobs) and `fastdist` [Boger, 2021] (fast distance calculations in Python). We ran the analysis code on a machine with two GPUs (GeForce RTX 2080 Ti $\times 2$), one CPU (Intel i9-7900X @ 3.30GHz) and 64 GB of RAM. The computer was running an Ubuntu 18.04.5 operating system. In practice, we were only allotted to use a subset of the resources, as it was a shared computer by the research group in machine learning at the University of Bergen. Finally, the code used to perform the analyses is publicly available via GitHub in [Triki, 2021].

3.1 Training and evaluation of word2vec

In this section, we will describe how we trained and evaluated our word2vec model. In particular, we will explain the data preprocessing choices we made before training word2vec in Section 3.1.1 and details of our implementation of word2vec using the Skip-gram model and negative sampling in Section 3.1.2. Finally, we will cover the hyperparameter choices used to train the word2vec model in Section 3.1.3 and evaluate the performance of the word2vec model using analogy test data sets in Section 3.1.4.

3.1.1 Data preprocessing

To train a word2vec model, we require a sufficiently large data set and embedding dimensionality to yield good quality word embeddings [Mikolov et al., 2013b]. In the empirical experiments of [Mikolov et al., 2013b], they used an internal data set based on data from Google News. Since this data set was not publicly available, we instead used a Wikipedia dump from [Wikimedia, 2021] (i.e. a periodic snapshot of the Wikipedia database), and we performed several preprocessing steps before training on it. In particular, we used the *enwiki* (short for English Wikipedia) dump from the 1st of January 2021 (20210101 on the Wikimedia pages). The dump from Wikipedia was first downloaded and parsed using the WikiExtractor tool [Attardi, 2015]. Furthermore, we created a script using Python to merge and process output files from the WikiExtractor tool into a certain number of text files, such that we could train word2vec at ease. To benefit from parallel reading, we let the number of text files equal the number of CPU cores on our machine.

We then proceeded by processing each Wikipedia article. In particular, we performed the following steps:

1. We split each article into a list of sentences using the `tokenize.sent_tokenize` function from the `nltk` Python package [Bird et al., 2009].

2. Then, we preprocessed each sentence individually.

   2.1. We first replaced contractions in each sentence (e.g. I'll $\mapsto$ I will, you'd $\mapsto$ you would, etc.) by using the `contractions` Python package [van Kooten, 2016].

   2.2. Then we split the sentence into a list of words using the `word_tokenize` function from `nltk`.

      i. We replaced capital letters in words by the corresponding small letters (i.e. lower-case representation).

      ii. We removed punctuation from words and created new sub-words for each word delimited by punctuation (e.g. out-of-the-box $\mapsto$ out, of, the, box).

      iii. At last, we replaced all numbers (including ordinal numbers) with their textual representation, using the `num2words` Python package [Dupras, 2014]. For example, the number 10 was replaced by "ten", and the word "21st" was replaced by "twenty-first".

3. With the new processed sentences, we filtered out sentences that had less than **min_word_count** words in them.

4. Finally, we appended each sentence to an output text file, separated using the newline character (i.e. \n).

After processing the Wikipedia articles into files, we combined common phrases into single tokens. In particular, we followed the word2phrase procedure explained in Section 2.2.2.4, resulting in tokens consisting of words separated by an underscore, e.g. the phrase "New York" becomes "new_york". We denoted the threshold hyperparameter from word2phrase as **threshold-word2phrase**. To create longer phrases of words, e.g. trigrams, four-grams or even five-grams, we repeated the word2phrase multiple times. In particular, we denote the number of repetitions as **num-epochs-word2phrase**, which we chose as a hyperparameter. Furthermore, for each repetition of word2phrase, the threshold hyperparameter δ is decreased. [Mikolov et al., 2013b] did not state how they decreased this threshold, however, but by inspection of the source code of word2vec [Mikolov, 2013a], we observed that they started with a threshold of 200, then decreased it to 100 for the second and final repetition.

With this in mind, we introduce a threshold decay hyperparameter, denoted **threshold-decay-word2phrase**, which tells how much the threshold decreases for each repetition of word2phrase.

3.1.2 Implementation specifics

To implement the word2vec model, we used Python and TensorFlow [Abadi et al., 2015]. In addition to this, we used the `numpy` [Harris et al., 2020] package to work with vectors and matrices more easily. In particular, we implemented the Skip-gram model using negative sampling. To do so, we split our implementation into three main Python classes. The first class is the `Tokenizer` class, which is responsible for converting text into word indices in vocabulary (e.g. the word "hello" $\mapsto$ 42). The second class is the `Word2vecSGNSModel`, which inherits the `tf.keras.Model` class from TensorFlow; we created the model via subclassing, as specified in [TensorFlow team, 2020]. `Word2vecSGNSModel` is the model we used to train our ANN. The third and final main class is `Word2vec`. It performs training using the `Word2vecSGNSModel` and uses `Tokenizer` to convert words into integers.

To load the data into the model, we used the `tf.data` API, as introduced in TensorFlow 2. The `tf.data` API allows us to create flexible and scalable data generators. As mentioned in Section 3.1.1, we want to train our model on dumps from Wikipedia, i.e. several gigabytes of raw text data, and the `tf.data` API allows us to do this quickly and efficiently. In particular, we used the `tf.data.TextLineDataset` class to load multiple text files in parallel and set `num_parallel_calls` to `tf.data.experimental.AUTOTUNE` wherever we could, such that we parallelize the data generation process as much as possible. We also used `prefetch` to prepare the data in parallel while training.

We implemented word2phrase using Python. First, we counted the uni- and bigram word occurrences, and using them, we ran the word2phrase procedure as explained in Section 2.2.2.4 by accepting bigrams into the vocabulary if the phrase score (see Equation (2.66)) was greater than the set threshold parameter.

By implementing word2vec ourselves, we learned a few things we did not realize after reading the two papers from Mikolov et al. [Mikolov et al., 2013a, Mikolov et al., 2013b]:

- Training on big data sets (e.g. dumps from Wikipedia) requires an efficient implementation of the data generator. We first attempted to create a data generator that loaded everything into memory, but it became clear to us that this did not scale well when we later wanted to train on bigger data sets.
- The quality of the word embeddings depend on the preprocessing of the training data.
- That we have two embedding matrices W and W' corresponding to the input and output of the network. At first, we only had a single embedding matrix, for both the input and the output of the network, which led to worse results.

3.1.3 Hyperparameter choices

To train the word2vec model, we based our choices of hyperparameters on the different choices used in models from [Mikolov et al., 2013a, Mikolov et al., 2013b]. These hyperparameters can be found in Table 3.1.

Hyperparameter	Value
min-word-count	5
max-vocab-size	∞
batch-size	256
num-epochs	5
num-epochs-word2phrase	2
threshold-word2phrase	200
threshold-decay-word2phrase	0.5
learning-rate	0.025
min-learning-rate	0.0000025
embedding-dim	300
max-window-size	5
num-negative-samples	5
sampling-factor	0.00001
unigram-exponent	0.75

Table 3.1: Hyperparameters used to train the word2vec model.

Similar to [Mikolov et al., 2013b], we set the minimum word count to 5 and did not restrict the maximum vocabulary size. In other words, we let the vocabulary include words that occur at least 5 times in the training data.

We set the number of repetitions for word2phrase to 2 and the initial threshold to 200, as [Mikolov et al., 2013b] did in their experiments. Furthermore, we set the threshold decay to 0.5 (i.e. the threshold is halved for each repetition) to use a similar setup.

Neither [Mikolov et al., 2013a] nor [Mikolov et al., 2013b] stated which batch-size they used, but by inspecting the source code of word2vec [Mikolov, 2013c, line 542], we observed that they used 1 as their batch size, i.e. performing a backward pass for every forward pass in the model. We found, however, that setting the batch size to 256 to be a nice fit for our data, leading to good quality vectors and faster training.

Mikolov et al. used 1 to 4 epochs in their experiments [Mikolov et al., 2013a, Mikolov et al., 2013b], and in the source code of word2vec [Mikolov, 2013c, line 43], they default to 5 epochs. For this reason, we set the number of epochs to 5.

We set the initial and minimum learning rate to 0.025 and 0.000025, respectively, as noted in [Mikolov et al., 2013a] and the source code of word2vec [Mikolov, 2013c, lines 44 and 398].

Furthermore, we set the embedding dimension to 300, the maximal window size to 5, the number of negative samples to 5, the sampling factor to 0.00001 and the unigram exponent to 0.75, similar to experiments from [Mikolov et al., 2013b].

Using the preprocessing steps from Section 3.1.1 on our data and the hyperparameters from Table 3.1, we get a vocabulary size of ∼4.4 million words and corpus size (i.e number of words used from the *enwiki* data set) of ∼2.3 billion words.

3.1.4 Model evaluation

We trained the word2vec model using data preprocessing steps from Section 3.1.1 and hyperparameters from Section 3.1.3. Following, we will refer to our trained word2vec model as the *SGNS-enwiki* (short for **S**kip-**G**ram **N**egative **S**ampling-enwiki) model. To show that the trained word embeddings from the SGNS-enwiki model can be used for word analogy tasks, we evaluated the SGNS-enwiki model using analogy test data sets. The goal of performing these tests is to show that the word embeddings of the SGNS-enwiki model are comparable to word embeddings from other published (pre-trained) models in terms of quality.

In particular, we used three analogy test data sets, namely the *Semantic-Syntactic Word Relationship test set* (SSWR), the *Microsoft Research Syntactic Analogies Dataset* (MSR) and the *Phrase Analogy Dataset* (PAD). The SSWR test data set was first introduced in [Mikolov et al., 2013a], consists of 8869 semantic and 10675 syntactic questions and is widely used as a test data set. The MSR data set was first introduced in [Mikolov et al., 2013c] and consists of 8000 analogy questions. To evaluate word embedding models trained on phrases (e.g. "New York Times"), [Mikolov et al., 2013b] introduced the PAD. PAD consists of 3218 analogy questions. It should be noted, however, that there are other common test data sets as well, such as the Bigger analogy test set (BATS) from [Gladkova et al., 2016].

We compared the results from the evaluation of the SGNS-enwiki model to models from [Mikolov et al., 2013a, Mikolov et al., 2013b, Mikolov et al., 2013c, Bojanowski et al., 2017] in Tables 3.2 to 3.4. In particular, we compared to the Skip-gram models from [Mikolov et al., 2013a, Table 3] and [Mikolov et al., 2013a, Table 6] (denoted *SG 300* and *SG 1000* respectively), the *NEG-15* model from [Mikolov et al., 2013b, Table 1 and 3], the *RNN-1600* model from [Mikolov et al., 2013c, Table 2], the *GloVe 300 42B* model from [Pennington et al., 2014, Table 2], and the *fastText* model from [Bojanowski et al., 2017, Table 2]. In Tables 3.2 to 3.4, a dash (–) denotes that the model has not been evaluated on the particular subset/data set, and **bold** values indicate the best value. Values represent accuracies and are in percentages.

Model	SSWR		
	Semantic	Syntactic	Average
SG 300	55	59	57
SG 1000	66.1	65.1	65.6
NEG-15	61	61	61
RNN-1600	–	–	–
GloVe 300 42B	**81.9**	69.3	75.0
fastText	77.8	**74.9**	**76**
SGNS-enwiki	65.8	67.3	66.6

Table 3.2: Comparison of empirical results of word embedding models evaluated using the SSWR word analogy test data set.

Model	MSR			
	Adjectives	Nouns	Verbs	Average
SG 300	–	–	–	**56**
SG 1000	–	–	–	–
NEG-15	–	–	–	–
RNN-1600	23.9	29.2	**62.2**	39.6
GloVe 300 42B	–	–	–	–
fastText	–	–	–	–
SGNS-enwiki	**43.1**	**62.5**	59.1	54.9

Table 3.3: Comparison of empirical results of word embedding models evaluated using the MSR word analogy test data set.

Model	PAD
	Average
SG 300	–
SG 1000	–
NEG-15	42
RNN-1600	–
GloVe 300 42B	–
fastText	–
SGNS-enwiki	**53.7**

Table 3.4: Comparison of empirical results of word embedding models evaluated using the PAD word analogy test data set.

In Table 3.2, we see that the SGNS-enwiki model is fairly competitive in terms of accuracy on the SSWR analogy test data set. The fastText model, however, is the most accurate model on this test data set. In particular, the fastText model is approximately 10% more accurate on average than the SGNS-enwiki model. The same story goes for the results from the MSR test data set, we see in Table 3.3, where the SGNS-enwiki model performs pretty well, falling short for the SG 300 on average. Finally, in Table 3.4 we see that the SGNS-enwiki model outperforms the NEG-15 model. We note that we had a lot of missing data for this evaluation, as all models had not been evaluated for every (subset of the) test data set. This evaluation, however, indicates that the SGNS-enwiki model understands syntactic and semantic relationships between words.

To gain further insight into how the vector representations learned by the SGNS-enwiki model are, we inspected the nearest neighbours of words. In Table 3.5 we show a sample of such comparison, using the 5 nearest neighbouring words (also some phrases) for each query word. We used cosine similarity to find the neighbouring words, excluding the query word from the search. In Table 3.5, we see the ability of the SGNS-enwiki model to identify related words to the query word.

Query word	Neighbouring words
Apple	Apple Inc., Blackberry, Apple computer, OneScanner, released Xsan
Phone	Phones, mobile phone, cell phone, cellphone, phone calls
Water	Fresh water, drinking water, water pumped, salinated, untreated water
Sunny	Windy, dry sunny, warm sunny, cool, Lee Hany Lee
Book	Books, book entitled, Tarcher Penguin, author, foreword

Table 3.5: The five nearest neighbouring words of some query words. We use cosine similarity and word embeddings of the SGNS-enwiki model.

We visualize the ability of the SGNS-enwiki model to identify underlying concepts of the language and relationships between them in Figure 3.1, using a 2-dimensional PCA (Section 2.1.6.1) embedding of words representing countries/capitals and comparative adjectives (e.g. good $\rightarrow$ better $\rightarrow$ best). We used PCA instead of UMAP here as there were few points, and PCA typically works better than UMAP in such cases. In Figure 3.1, we see that the SGNS-enwiki models understand what capital means and how comparative adjectives behave. In addition to this, we also observe some clustering occurring in both plots. In particular, we observe that Scandinavian countries and capitals are more clustered to the top of Figure 3.1 (a), and words related to temperatures are more clustered to the right of Figure 3.1 (b).

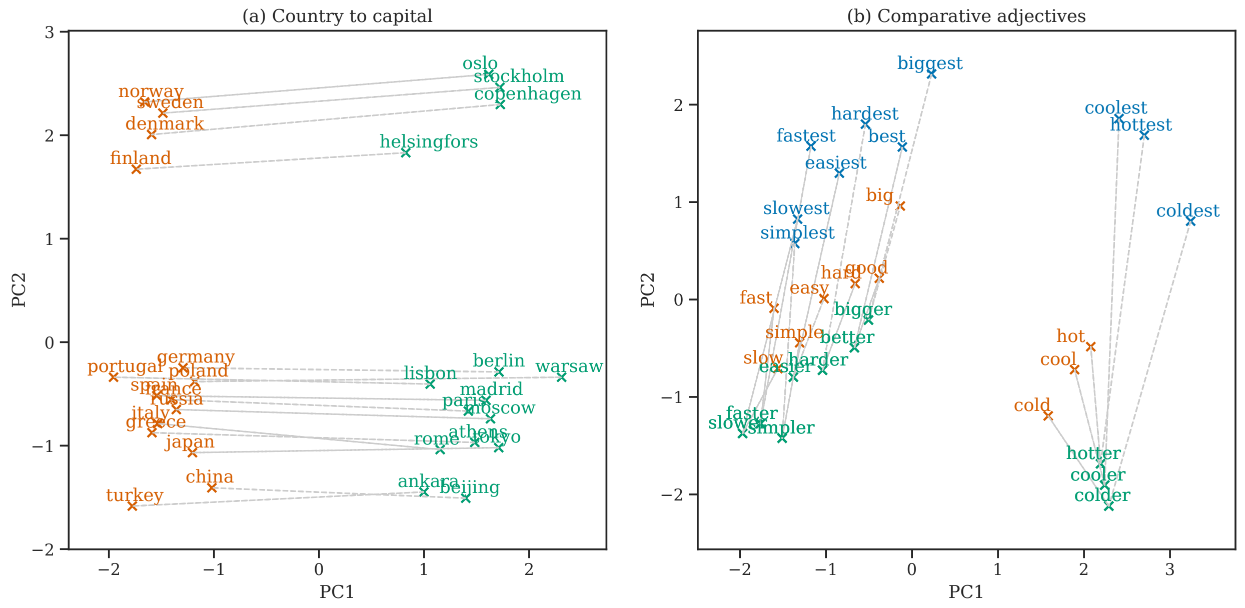

Figure 3.1: 2-dimensional PCA embeddings of the word embeddings of the SGNS-enwiki model. The plots show how the SGNS-enwiki model understands concepts such as countries and their capital cities (a), as well as and comparative adjectives (b). This figure is inspired by [Mikolov et al., 2013b, Figure 2].

Next, we will further investigate the notion of clustering, partially motivated by the results we see in Figure 3.1. In particular, to deepen our understanding of the underlying structure of the SGNS-model, we will in the next section perform cluster analysis of its word embeddings. We will use multiple clustering algorithms and internal cluster validation methods to find the most suitable clustering algorithm and hyperparameters.

3.2 Word clustering

In this section, we will apply cluster analysis on the word embeddings of the SGNS-enwiki, to search for deeper structures within the data. In particular, we will compare clustering algorithms on the word embeddings of the SGNS-enwiki in Section 3.2.1, and following, we will look at clustering of distinct groups of words in Section 3.2.2.

3.2.1 Comparing clustering algorithms

In this subsection, we compare clustering algorithms on the word embeddings of the SGNS-enwiki. Due to a large number of words in the vocabulary of the SGNS-enwiki (roughly 4.4 million, see Section 3.1.3 for more details), we restrict the analysis to the 10000 most common (i.e most frequently occurring) words. This way, we speed up the computation by reducing the computational requirement. Also, we note that we should still get reliable results, as the most common words yield good quality vector representations due to the nature of their word frequencies.

To perform the cluster analysis, we used all clustering algorithm from Section 2.1.2, except for Spectral clustering (Section 2.1.2.6), as it was too computationally expensive to run. In particular, we used the following algorithms: k-means clustering (Section 2.1.2.1), mini-batch (MB) k-means clustering (Section 2.1.2.2), k-medoids clustering (Section 2.1.2.3), GMMs (Section 2.1.2.4), hierarchical clustering (agglomerative) (Section 2.1.2.5), HDB-SCAN (Section 2.1.2.7) and ToMATo (Section 2.1.2.8). We used the `scikit-learn` [Pedregosa et al., 2011], `scikit-learn-extra` [Scikit-learn contrib, 2021], `gudhi` [Rouvreau, 2021] and `hdbscan` [McInnes et al., 2017] Python packages to perform clustering. Furthermore, we trained the clustering algorithms using a grid-search manner, i.e. by trying all combinations of hyperparameters. Table 3.6 shows the hyperparameters used to train each clustering algorithm. By forming a grid of hyperparameters for each clustering algorithm, we get a rough sense of the best set of hyperparameters. For the initial grid-search, we let `n_clusters_range`=2, 3, 4, 5, 10, 50, 100, 150, 200, 300, 400, 500, 750, 1000, 1500, 2000, 3000, 4000, 5000, 6000, 7000, 8000 be the cluster numbers used in algorithms where applicable. We let `n_clusters_range` range from 2 to 8000 clusters, using varying step sizes, to investigate the effect of the number of clusters for each algorithm, where it was applicable. To train the clustering algorithms, we used the standard word embeddings if the algorithm supported cosine similarity (or distance) and normalized word embeddings if the algorithm required Euclidean distances. After training the clustering algorithms, we validated them using the internal cluster validation methods from Section 2.1.3. In particular, we used the mean Silhouette Coefficient (SC) (Section 2.1.3.1), the Davies-Bouldin Index (DBI) (Section 2.1.3.2) and the Caliński-Harabasz Index (CHI) (Section 2.1.3.3). We used the `scikit-learn` Python package to perform internal clustering validation.

Clustering algorithm	Hyperparameters	Values
K-means clustering	`n_clusters`	`n_clusters_range`
Mini-batch k-means clustering	`n_clusters`	`n_clusters_range`
	`batch_size`	100
K-medoids clustering	`n_clusters`	`n_clusters_range`
GMM clustering	`n_components`	`n_clusters_range`
Agglomerative clustering	`n_clusters`	`n_clusters_range`
	`linkage`	`single`, `average`, `complete`, `ward`
HDBSCAN	`min_cluster_size`	2, 4, 8, 16, 32, 64
	`min_samples`	1, 2, 4, 8, 16, 32, 64
ToMATo	`density_type`	`DTM`, `logDTM`, `KDE`, `logKDE`
	`k`	2, 3, . . . , 10, 20, . . . , 50, 100, . . . , 250

Table 3.6: Hyperparameters used to train each clustering algorithm for the cluster analysis.

We visualize the result from the initial grid-search in Figure 3.2, where we see that the agglomerative clustering algorithm performs the best (close to k-means clustering) and k-medoids clustering performs the worst. For this reason, we will now focus on the agglomerative clustering algorithm and search for the best set of hyperparameters. Particularly, we will search for the best linkage criterion and number of clusters.

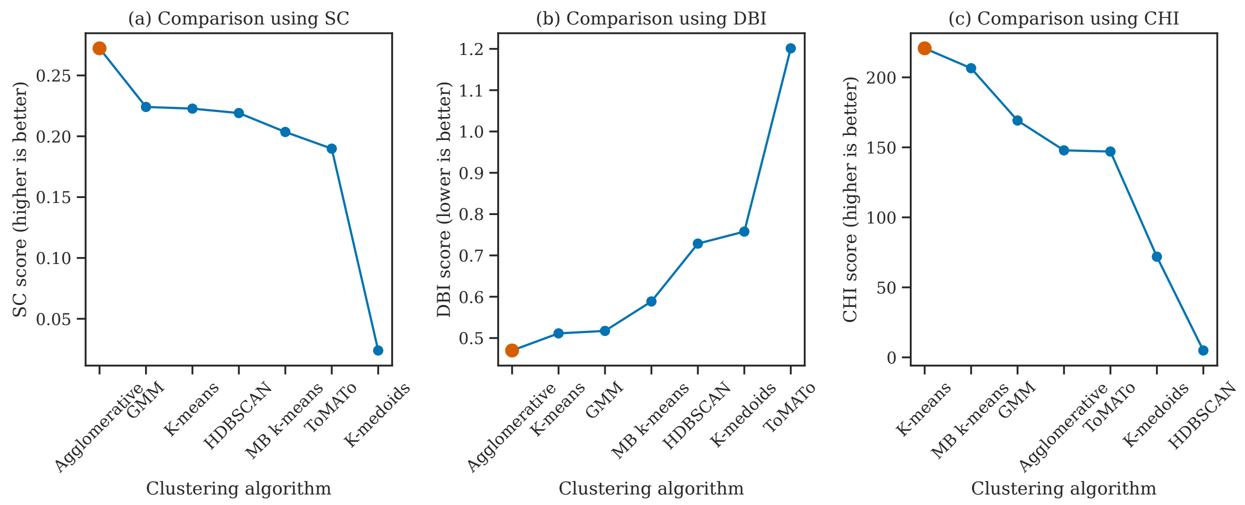

Figure 3.2: Comparison of internal cluster validation results from clustering algorithms trained on word embeddings of the SGNS-enwiki model. The red dot in each plot denotes the most optimal value.

We will now look at the initial grid search result using the agglomerative clustering algorithm to deepen our understanding of the results. We visualize the results in Figure 3.3, and we notice that by using the single linkage criterion, we get relatively poor results. For the remaining criteria, we observe that we get more or less the same results, with the ward criterion being slightly ahead of the rest. By inspecting the best value for the number of clusters for each internal cluster validation method in Figure 3.3, we observe that the DBI and the CHI gives misleading results, while the SC is more meaningful. In particular, the DBI prefers to have the largest number of clusters, i.e. 8000. We inspected the words falling into the clusters, and from these, we observed that 6350 of the words were in clusters of size 1. This result indicates that the DBI is not particularly well suited for choosing the number of clusters, as it prefers to have clusters consisting of exactly one word. Using the CHI, we observe that it prefers to have the least number of clusters, i.e. 2. We inspected this result and noticed that in the first cluster, there was only a single word, while the second cluster had the remaining 9999 words. In other words, this indicates that the CHI is also not particularly well suited for choosing the number of clusters. Finally, using the SC, which we show in Figure 3.3 (a), we observe that the preferred number of clusters lie around 3000 to 6000. We inspected the number of clusters as preferred by average, complete and ward linkage clustering and observed that they made sense, as there was more variety in the cluster sizes and the number of clusters having the specific cluster sizes. This indicates that the most preferable number of clusters (using SC) should lie in this range (3000 to 6000), and following, we will narrow down the search for the best number of clusters. For the next experiment, we will not include the single linkage clustering criterion, as it performed poorly.

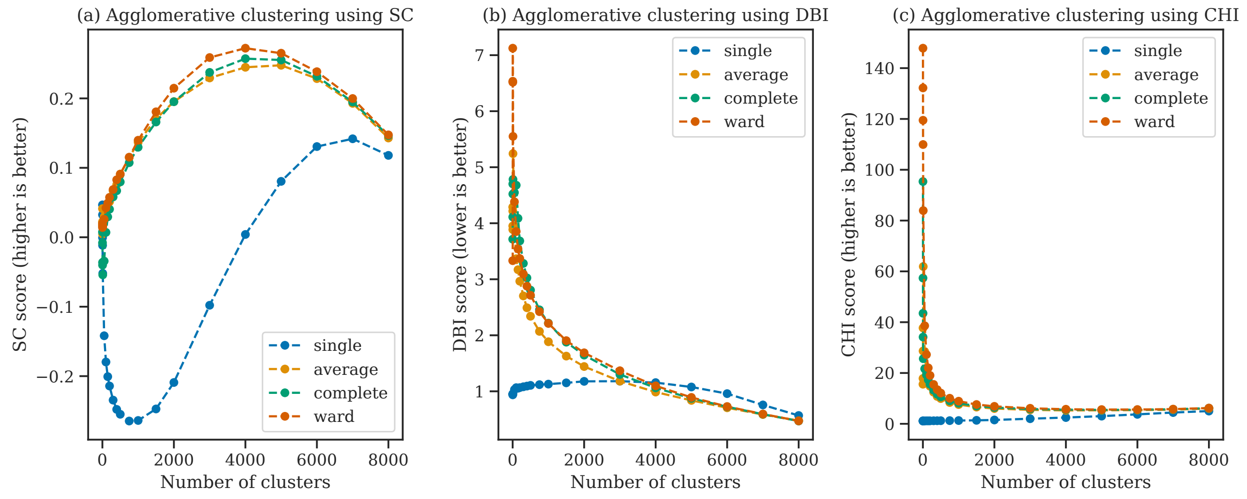

Figure 3.3: Internal cluster validation results using agglomerative clustering on word embeddings of the SGNS-enwiki model.

By narrowing the search to the range of 3000 to 6000 clusters, we find the best number of clusters for each criterion of agglomerative clustering. The narrowed search for the number of clusters is visualized in Figure 3.4, and we observe that ward linkage clustering with 4104 clusters results in the best clustering. In other words, these results indicate that, by using the ward clustering criterion, we obtain the best clustering of the 10000 most common words from the SGNS-enwiki model.

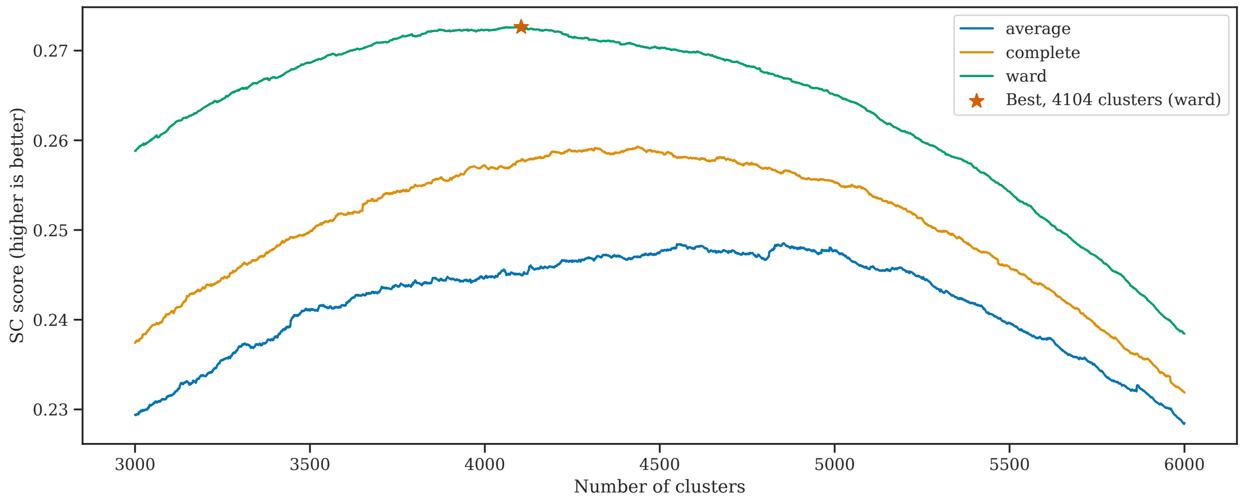

Figure 3.4: The number of clusters plotted against the SC scores. The number of clusters is in the range of 3000 to 6000 clusters, and we use the average-, complete- and ward-linkage criteria. Here we see that the ward linkage criterion results in the highest SC score.

To deepen our understanding of the clustering result using agglomerative clustering and ward criterion on the word embeddings from SGNS-enwiki, we investigated the words falling into the 4104 clusters. In particular, we looked at the 10 largest and smallest clusters. We restricted the smallest clusters to contain at least 2 words, ensuring that we do not get clusters consisting of single words. In the top 10 largest clusters, we mostly saw names such as "Smith", "Wilson", or "Taylor" clustered together. We also saw the clustering of words representing numbers, e.g. "forty-five", "thirty-two" or "fifty-one", and the clustering of family-related words, e.g. "father", "son" and "brother". The top 10 smallest clusters mostly consisted of words that were strongly related to one another, such as "Adam" and "Noah", "card" and "cards", or "interior" and "exterior". We visualize some of the largest and smallest clusters in Figure 3.5, using a 2-dimensional UMAP (Section 2.1.4.2) embedding. To create the UMAP embedding, we used the `umap-learn` Python package [McInnes et al., 2018] and the default hyperparameterization. In Figure 3.5, we see that the clusters are

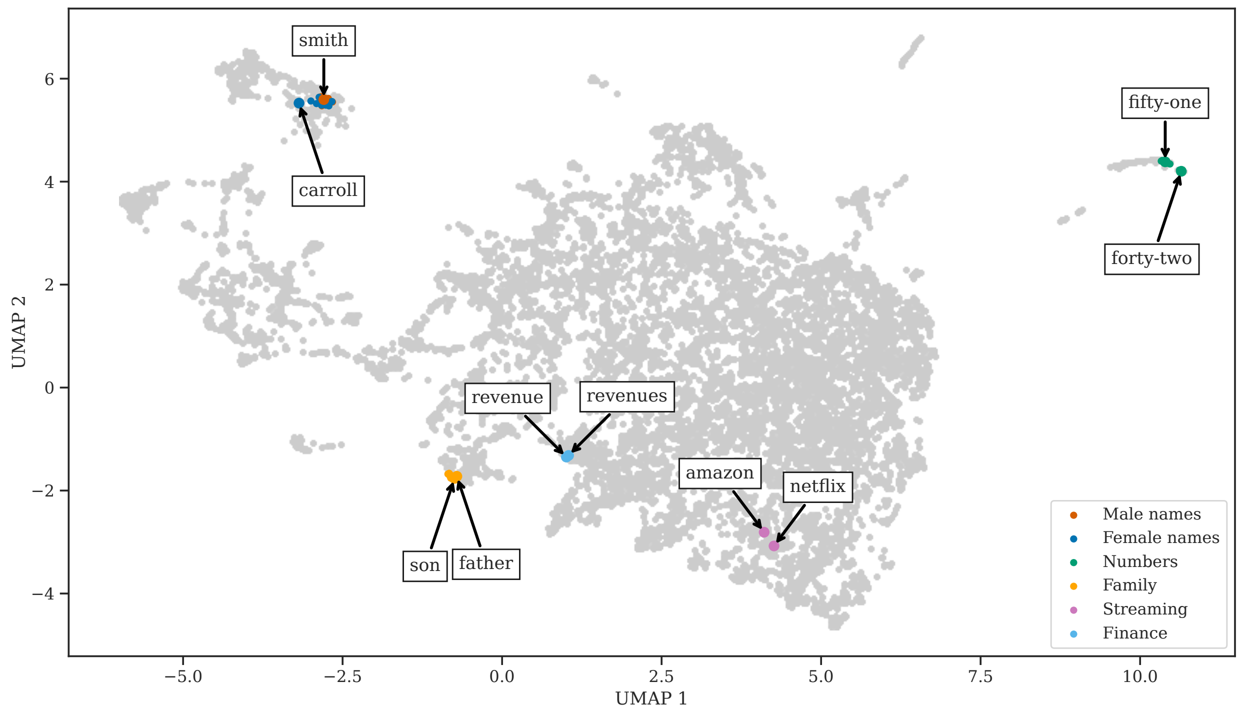

Figure 3.5: 2-dimensional UMAP embedding of the 10000 most common words from the SGNS-enwiki model, with some of the largest and smallest clusters outlined.

widely spread all over the UMAP embedding. In addition to this, the UMAP embedding suggests that there are more clusters throughout the word embeddings, which the clustering algorithms were unable to pick up when evaluated using internal cluster validation methods. We will investigate this further, and in the next subsection, we will look at the clustering of distinct word groups. In particular, we will see if bigger sets of words cluster together in the UMAP embedding, suggesting that the word embeddings contains a deeper structure.

3.2.2 Clustering word groups

In this subsection, we will investigate the effect of clustering in the 2-dimensional UMAP embedding of the 10000 most common words of the SGNS-enwiki model, using distinct groups of words. In particular, we will cluster words related to countries/capitals, numbers, names (forenames and surnames) and food. Before performing the clustering, we first prepare the data used for the analysis. The countries/capitals data was retrieved from [GeoNames, 2005], where we used their API to fetch countries and their capital, resulting in 217 pairs of countries and capitals. The number data was generated by converting numbers to their string representation. We converted the numbers from zero to one trillion, resulting in 105 number-related words. The forenames data was retrieved from [Social Security Administration, 2019], where we used the top 1000 baby names from 2019. The surnames data was retrieved from [U.S. Census Bureau, 2010], and we used the top 1000 surnames from 2010. Finally, the food data was retrieved from [Datafiniti, 2017], where we used the 250 most common ingredient words. We visualize the largest word group clusters falling into the 10000 most common words of the word embeddings from the SGNS-enwiki in Figure 3.6, where we use a 2-dimensional UMAP embedding. In Figure 3.6, we observe that two well-separated clusters are forming in the UMAP embedding, being the names and numbers word groups. We also see that the countries and food groups are more spread out in the embedding.

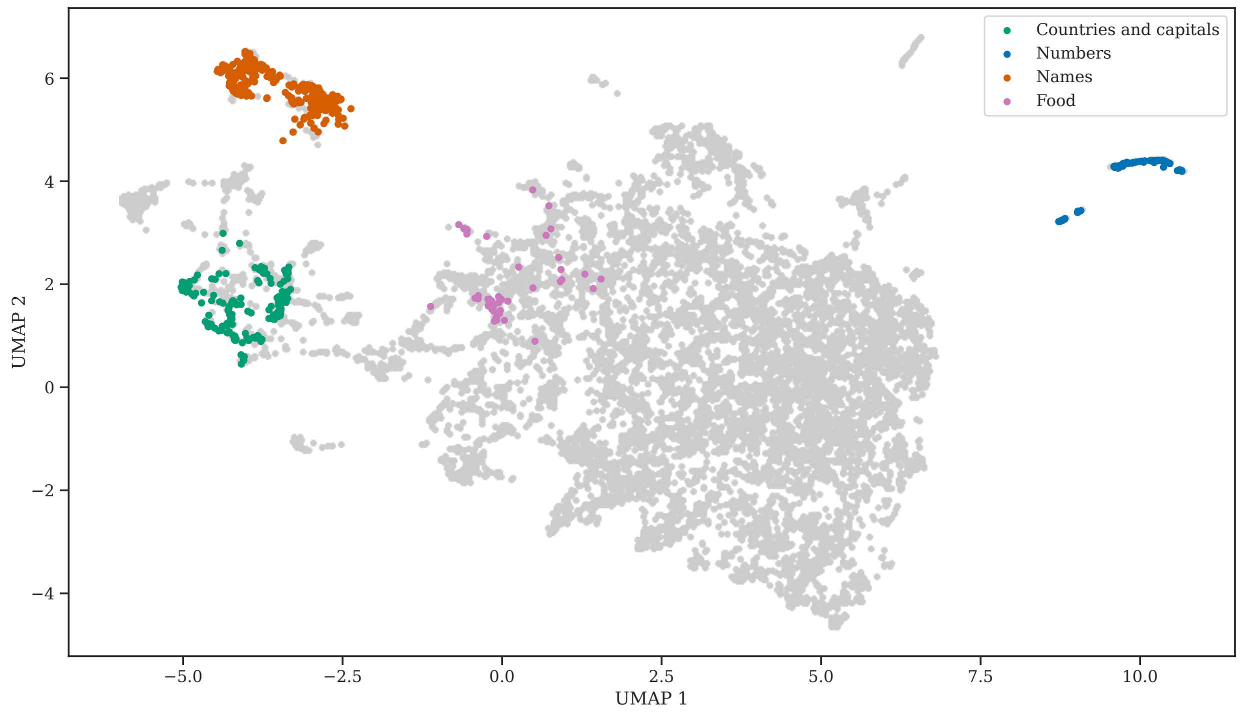

Figure 3.6: 2-dimensional UMAP embedding of the 10000 most common words from the SGNS-enwiki model, with word groups outlined.

In Figure 3.6, we outline the largest clusters of the word groups and discard words falling out of the largest clusters. By including words that are outside the largest clusters, we saw that, in particular, the names word group is spread throughout the word embedding, as the data we used contained forenames and surnames of common words, such as ”joy”,

”page” or ”good”. We illustrate this behaviour in Figure 3.7, where we outline the four different word groups. In Figure 3.7 (a), we see that the country and capital words are mostly clustered to the middle left, with some capitals falling out of the bigger cluster. The ”Stanley” and ”Hamilton” capital cities are also used as names, as indicated in Figure 3.7 (c). For the numbers, we observe that most number-related words are clustered to the right, clearly separated from the rest of the words. However, we also observe that words such as ”million”, ”billion” and ”trillion” are clustered together outside the numbers cluster to the right. By inspection, we observed that the ”million”, ”billion” and ”trillion” words were, in fact, close to other financial words, such as ”banks”, ”wealth” or ”economics”. In Figure 3.7 (c), we see that the forenames and surnames are clustered to the top left, in addition to being spread throughout the UMAP embedding. We also observe a small cluster of women names forming, containing the names ”Diana” and ”Isabella”. Finally, we see that food-related words in Figure 3.7 (d) are slightly clustered around the words ”egg” and ”cheese”, but also spread around the UMAP embedding. An interesting observation is the word ”apple”, which is both a fruit and a technology company. In this case, the word apple refers to the company Apple Inc., as we also saw earlier in Table 3.5.

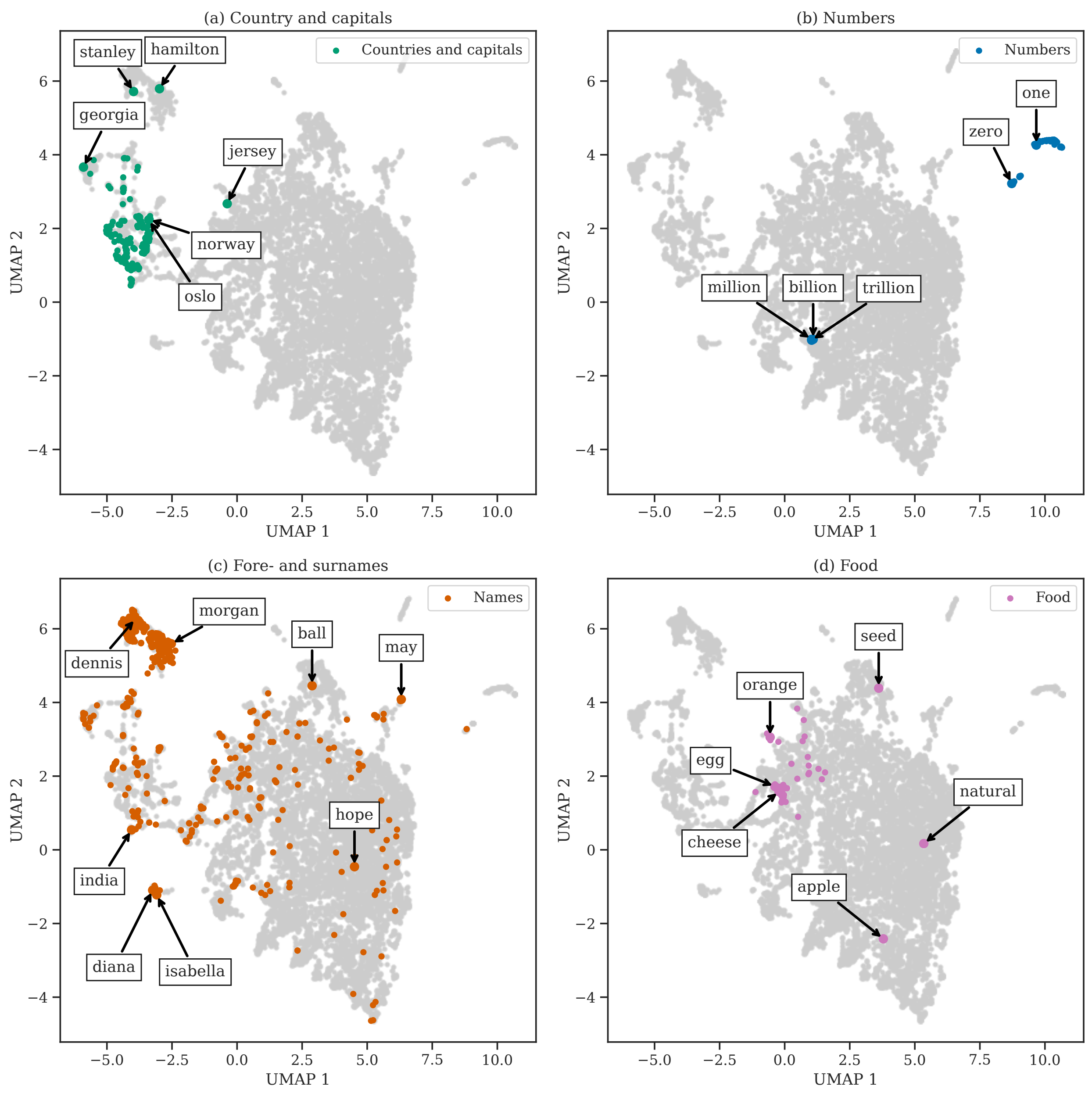

Figure 3.7: 2-dimensional UMAP embeddings of the 10000 most common words of the SGNS-enwiki model. Here we see four plots, and for each of them, we outline the four different word groups.

To further develop our understanding of the SGNS-enwiki word embeddings, we will analyze two of the previous word groups. In particular, we will perform cluster analysis of the word embeddings of countries/capitals and numbers, where we will use clustering algorithms to cluster the words. We will use the same clustering algorithms specified in Section 3.2.1, in addition to Spectral clustering. To visualize the results, we will use dimensionality reduction algorithms to create 2-dimensional embeddings. We will also use latitude/longitude coordinates of countries to visualize the clustering results using countries/capitals word embeddings.

We analyzed the countries and capital word groups separately, as we choose to either identify a country by its name or its capital. Starting with the country word group, we performed cluster analysis. The result of the cluster analysis is summarized in Figure 3.8, where we see a similar result to the result in Figure 3.2, i.e. agglomerative clustering is the preferred clustering algorithm.

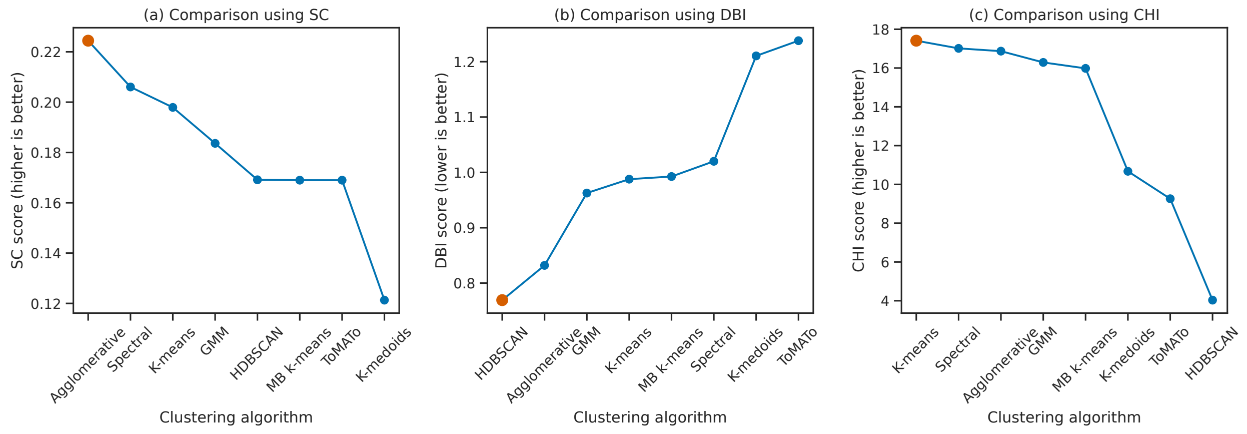

Figure 3.8: Comparison of internal cluster validation results from clustering algorithms trained on country word embeddings from the SGNS-enwiki model. The red dot in each plot denotes the most optimal value.

Following, we inspected the scores from the DBI and CHI methods. We observed a similar pattern to the analysis from Section 3.2.1, namely that DBI prefers every word to be in its cluster and CHI prefers to have the smallest number of clusters (i.e. 2). For this reason, we will mainly focus on the results using SC. Using agglomerative clustering, we visualize its result in Figure 3.9. In Figure 3.9, we see similar results to Figure 3.3, namely that ward criterion gives the best clustering when using agglomerative clustering.

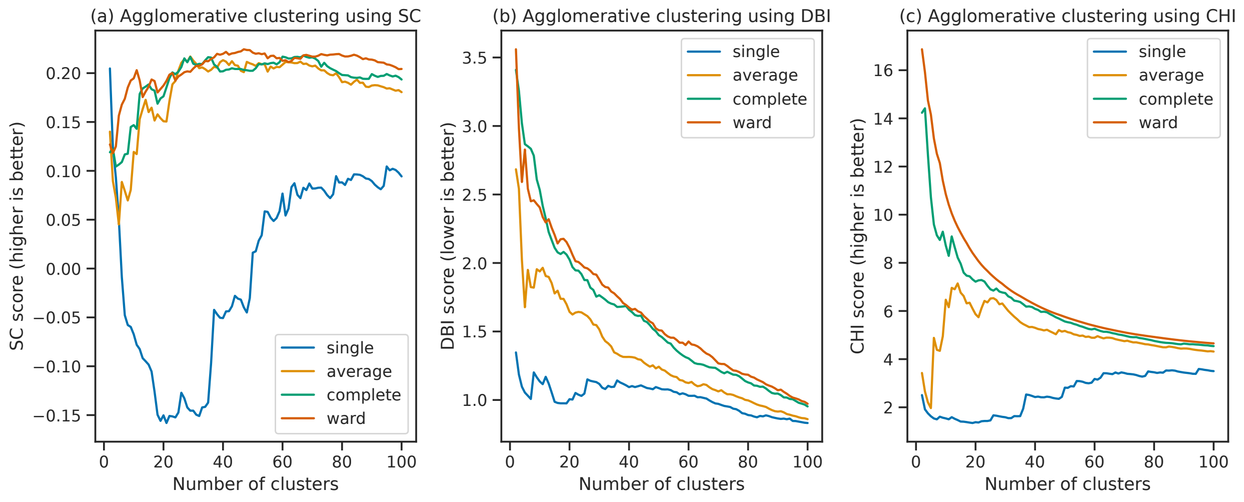

Figure 3.9: Internal cluster validation results using agglomerative clustering on country word embeddings of the SGNS-enwiki model.

The best clustering using SC with agglomerative clustering and ward criterion resulted in 47 clusters. We visualize this result using latitude/longitude coordinates of each country in Figure 3.10, where we see that the five largest clusters correspond well with the continent of the countries.

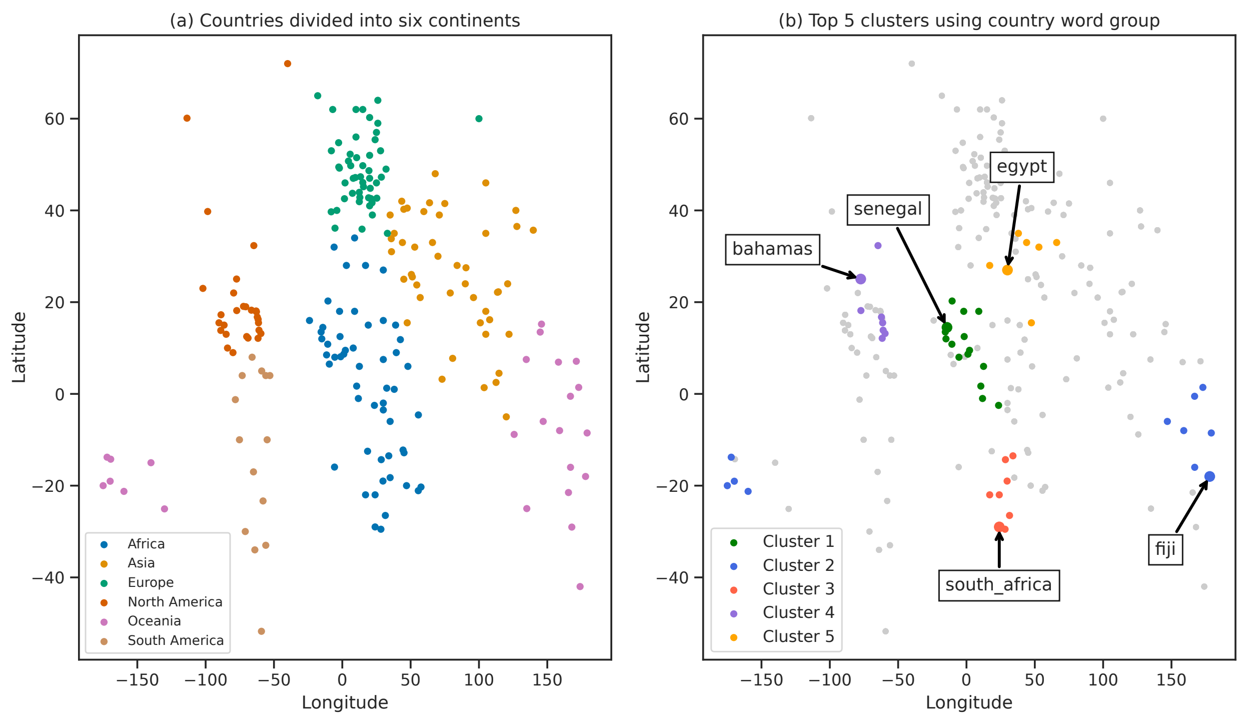

Figure 3.10: A comparison of countries divided into six continents (a) and the top 5 largest clusters from clustering of country word embeddings of the SGNS-enwiki model, using agglomerative clustering and ward criterion. Here we can see that the top 5 largest clusters in (b) correlate well with the continent of the respective countries.

Furthermore, we repeat the cluster analysis using capital to identify each country, i.e. we use the word embeddings of the capital words instead of the previously used country word embeddings. We show the result of the cluster analysis in Figure 3.11, where we see a similar result to the results in Figures 3.2 and 3.8, namely that agglomerative clustering is the preferred choice of clustering algorithm.

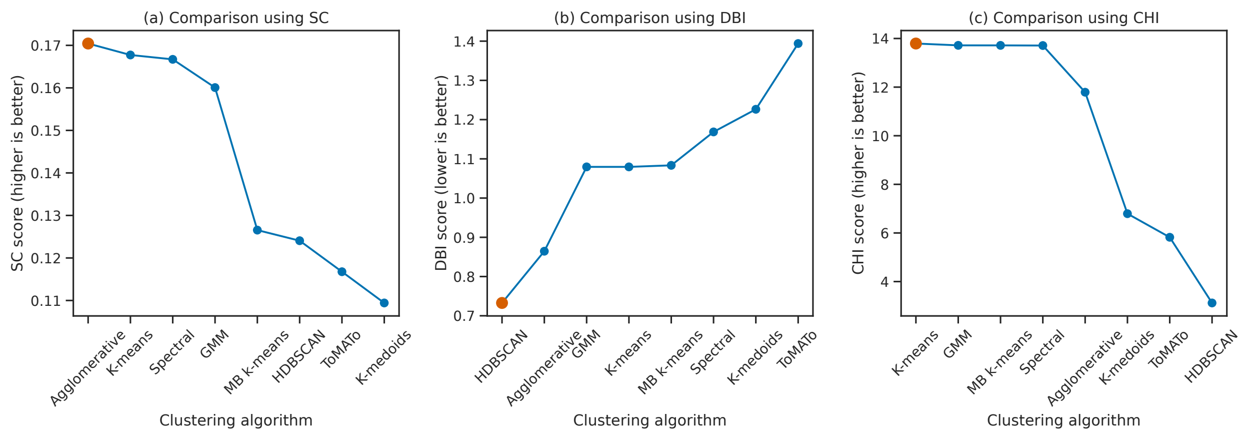

Figure 3.11: Comparison of internal cluster validation results from clustering algorithms trained on capital word embeddings from the SGNS-enwiki model. The red dot in each plot denotes the most optimal value.

We inspected the scores from the DBI and CHI methods, and similar to the results from Section 3.2.1 and the cluster analysis using country word embeddings, we saw that DBI prefers every word to be in its own cluster and CHI prefers to have the smallest number of clusters (i.e. 2). This further strengthens the motivation to use SC over the other methods, and we mainly focus on the results using SC. Using agglomerative clustering, we visualize the results using capital word embeddings in Figure 3.12, where we see similar results to Figures 3.3 and 3.9, namely that ward criterion gives the best clustering when using agglomerative clustering.

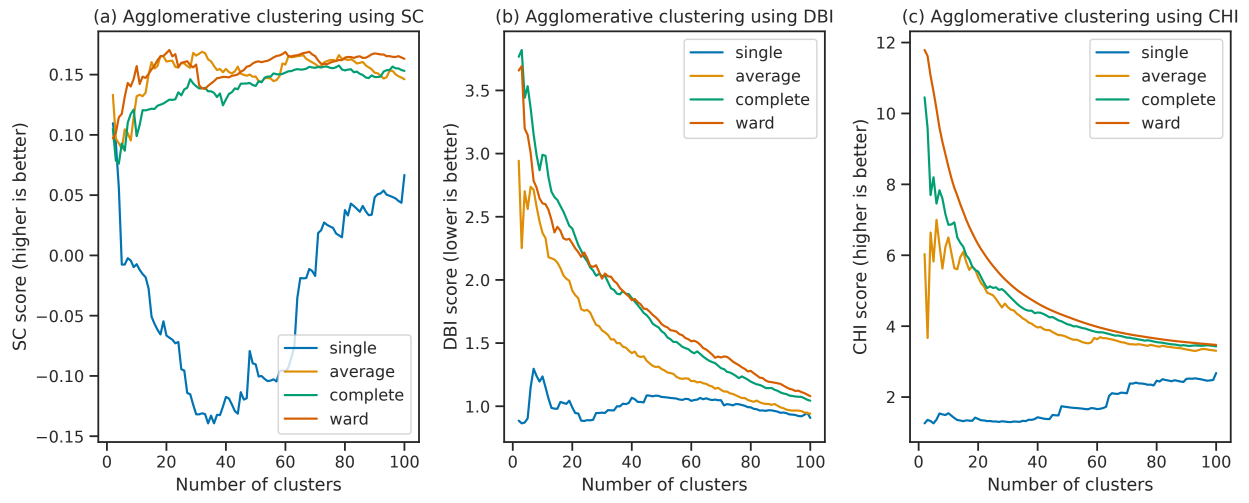

Figure 3.12: Internal cluster validation results using agglomerative clustering on capital word embeddings of the SGNS-enwiki model.

The best clustering using SC with agglomerative clustering and ward criterion resulted in 21 clusters. We visualize this result using latitude/longitude coordinates of each country in Figure 3.13, where we see that we get larger clusters than by using country word embeddings in Figure 3.10. Furthermore, we observe that in Figure 3.13 (b), the first cluster (green) consists of capitals where the countries are Spanish talking, as outlined by the "Madrid" (Spain), "Mexico City" (Mexico) and "Santiago" (Chile) boxes. The second cluster (blue) in Figure 3.13 (b) also correlates well with the Oceanic continent in Figure 3.13 (a), while the third (red) and forth (purple) clusters in Figure 3.13 (b) seem to capture the African continent adequately. Finally, we see that the (yellow) cluster consists of capitals from Eastern Europe and some capitals from Asia. For the record, Dakar is the capital of Senegal, Pretoria is one of the capitals of South Africa, and Suva is the capital of Fiji.

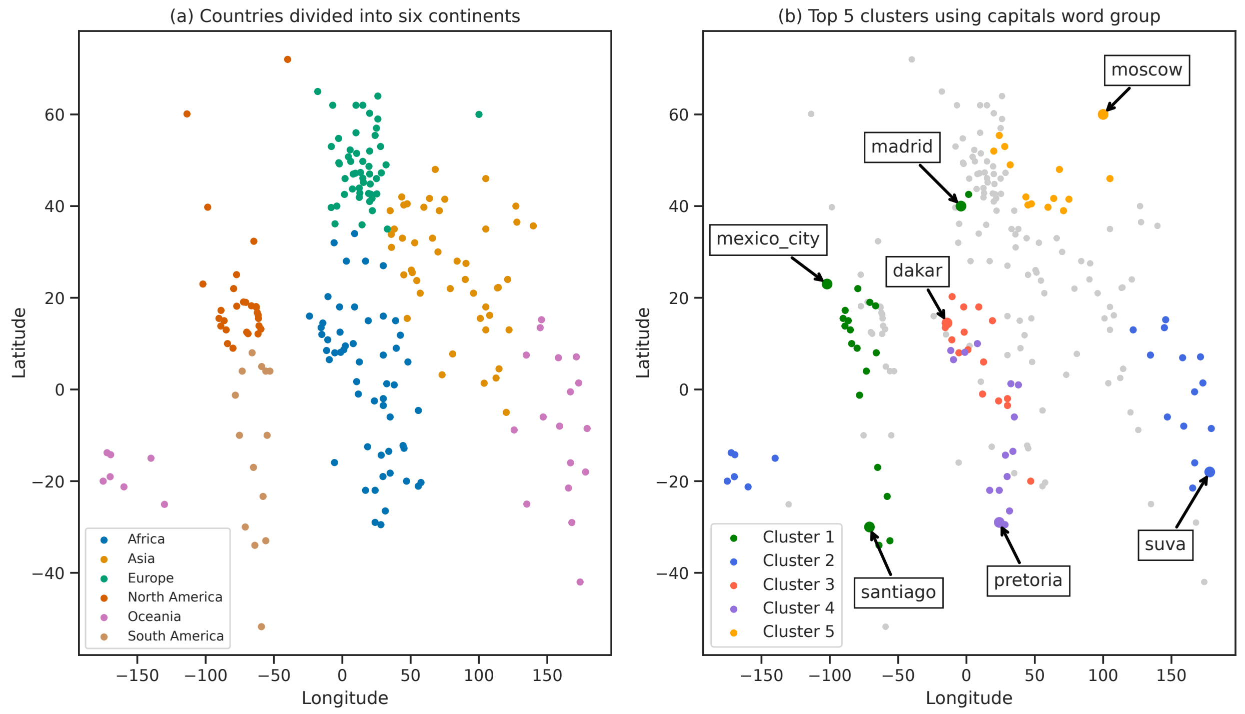

Figure 3.13: A comparison of countries divided into six continents (a) and the top 5 largest clusters from clustering of capital word embeddings of the SGNS-enwiki model, using agglomerative clustering and ward criterion. In (b) we can see that Spanish speaking countries are clustered together in the first cluster (green), while the other clusters are well clustered with respect to their continents.

Next, we will perform cluster analysis of number word embeddings in a similar manner to how we performed the cluster analysis of country/capital word embeddings. First, we show a comparison of clustering algorithms using internal cluster validation methods in Figure 3.14, where we see that, overall, the agglomerative clustering algorithm performs the best, when evaluated using internal validation methods.

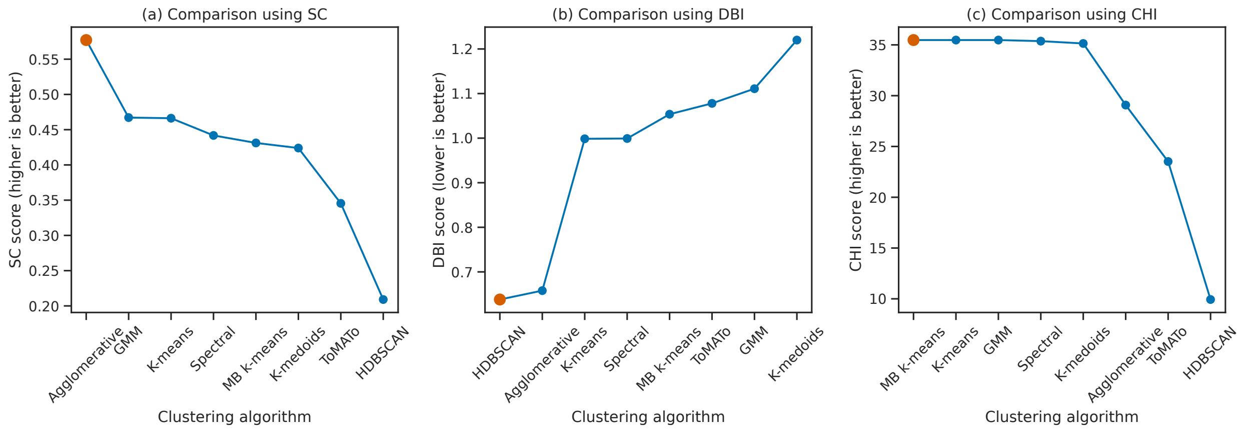

Figure 3.14: Comparison of internal cluster validation results from clustering algorithms trained on number word embeddings from the SGNS-enwiki model. The red dot in each plot denotes the most optimal value.

Furthermore, we will use the agglomerative clustering algorithm to cluster number word embeddings. To find its best criterion and number of clusters, we first visualize its results in Figure 3.15. In Figure 3.15, we see the internal validation methods prefer different amount of clusters and linkage criteria. In particular, SC prefers complete linkage criterion with 2 clusters, DBI prefers single linkage criterion with 6 clusters, and CHI prefers ward linkage criterion with 3 clusters. In other words, we observe a different behaviour than in the results from the internal cluster validation methods in Section 3.2.1 and the country/capital cluster analysis, namely that SC prefers the least amount of clusters, DBI does not prefer the most amount of clusters and CHI does not prefer the least amount of clusters.

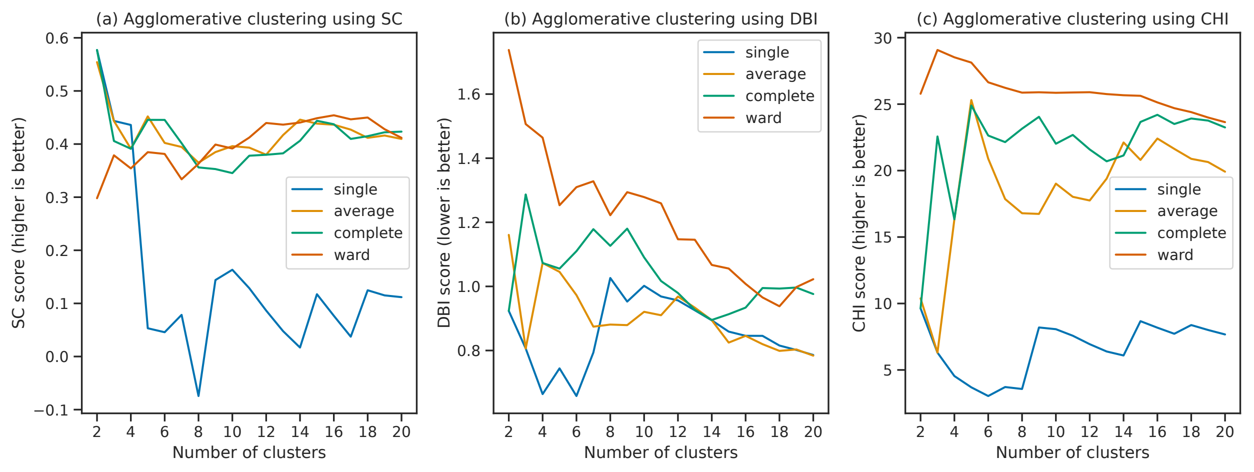

Figure 3.15: Internal cluster validation results using agglomerative clustering on number word embeddings of the SGNS-enwiki model.

Next, we visualize the best agglomerative clustering result of the number word embeddings as ranked by the internal clustering validation methods in Figure 3.16, where we use a 2-dimensional PCA embedding. We used PCA instead of UMAP here as there were few points to embed in 2 dimensions. In Figure 3.16, we see that it is not entirely clear how to cluster the number word embeddings.

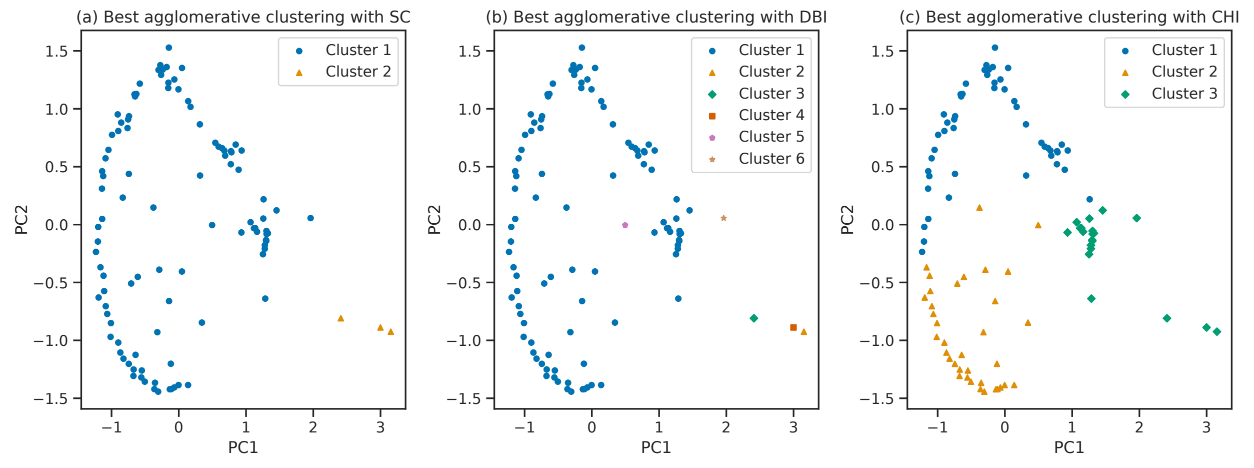

Figure 3.16: Comparison of the best clustering results using agglomerative clustering on the number word embeddings of the SGNS-enwiki model. Here we see that it is not clear which clustering is the best.

We further investigated the structure of the 2-dimensional PCA embedding of the number words and noticed an interesting relationship. We illustrate this relationship in Figure 3.17, which shows that by assigning an increasing label from the smallest and to the largest number, we get that the colour of the label gradually increases from the smallest to the largest. In other words, there seems to be an underlying sequential relationship in the number word embeddings. Furthermore, this suggests that the underlying structure of number word embeddings may contain information that we have not been able to find yet.

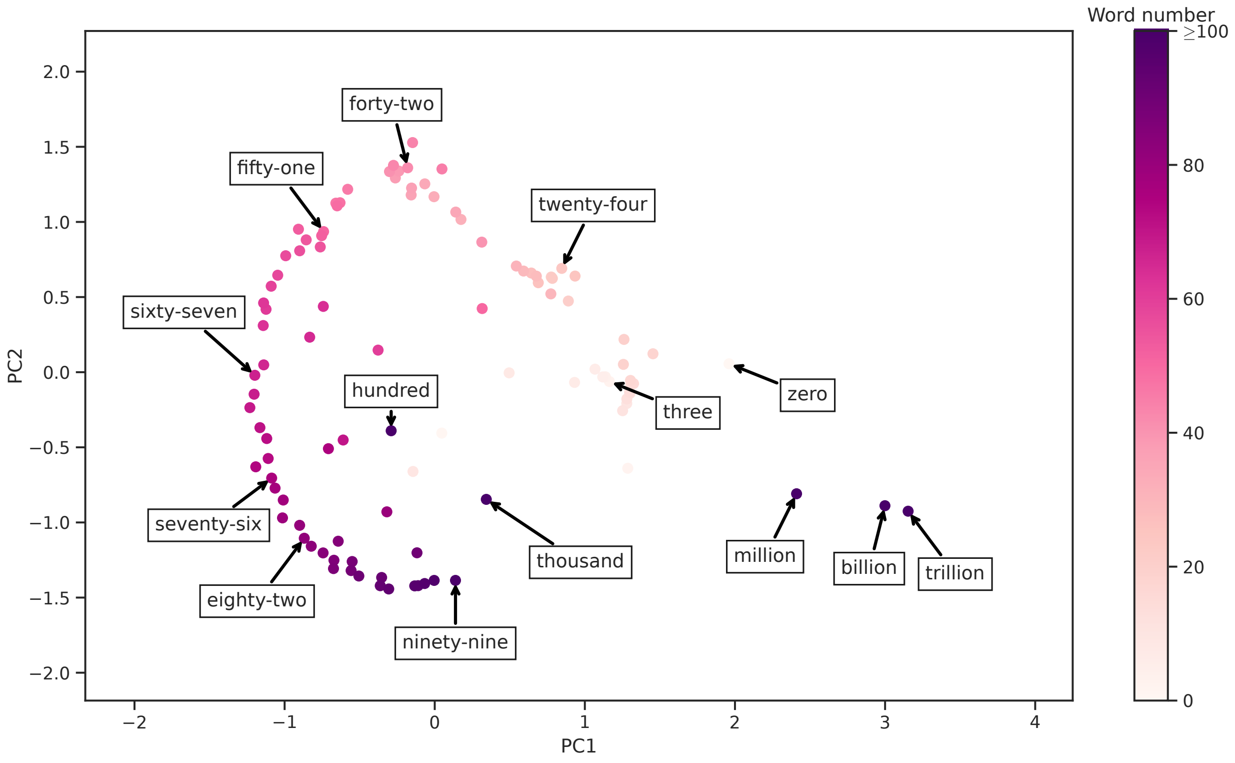

Figure 3.17: 2-dimensional PCA embedding of the 105 number word embeddings, where each word embedding has an increasing label assigned to it. Here we see that as we increase the number, we see a possible underlying sequential relationship.

3.3 Polysemous words prediction

In this section, we look at various methods for estimating the number of word meanings and predicting whether or not a word is polysemous. In particular, we will investigate the notion of topological polysemy in Section 3.3.1 and Geometric Anomaly Detection (GAD) in Section 3.3.2. We will use topological polysemy to estimate the number of word meanings of word embeddings. Particularly, we would like to see if the $\text{TPS}_n(w)$ score measures polysemy. Following this, we would like to see if singular word embeddings identified by GAD are polysemous. Next, we will compute the estimated intrinsic dimension of word embeddings and compare the results with the number of word meanings in Section 3.3.3. Finally, we end the section by proposing supervised models for predicting the number of word meanings and whether or not a word is polysemous in Section 3.3.4.

3.3.1 Topological polysemy

In this subsection, we will apply topological polysemy (Section 2.3.5) to the word embeddings of the SGNS-enwiki model. Additionally, we will also train a word2vec model using the same training data as in [Jakubowski et al., 2020], which we refer to as the *SGNS-semeval* model. We will apply topological polysemy to its word embeddings. Furthermore, we will compare the results to topological polysemy applied to word embeddings from pre-trained models, namely the fastText model (*fastText.TPS.300d*) used in experiments of [Jakubowski et al., 2020], the *GoogleNews-vectors-negative300* (shortened to *GoogleNews300*) word embeddings from [Google Code Archive, 2013], the *glove.840B.300d* word embeddings from [Pennington, 2014] and the English (*fastText.en.300d*) word embeddings from [Grave et al., 2018]. The fastText.TPS.300d model was kindly given in private communication with one of the authors of topological polysemy [Zibrowius, 2021].

The authors of topological polysemy, [Jakubowski et al., 2020], trained a fastText model on training data from the *SemEval-2010 Task 14: Evaluation Setting for Word Sense Induction & Disambiguation Systems* [Manandhar and Klapaftis, 2009]. The training data from the SemEval task consists of several sentences related to 100 polysemous words (50 nouns and 50 verbs). The SemEval data set also includes the number of true meanings (also called *gold standard* or *GS*) for each of the 100 polysemous words, as perceived by humans. In private communication with one of the authors of the topological polysemy measure [Zibrowius, 2021], we received some additional information regarding their training and data preprocessing choices. In particular, they stated that they used a fastText model with vector dimensionality of 300, that they removed all punctuation from words and replaced capital letters with the corresponding small letters. To compare with the $\mathrm{TPS}_n(w)$, the authors use the 100 polysemous words, words from the SemEval training data that has a *WordNet* [Fellbaum, 1998] entry and all words in SemEval training data. WordNet is a lexical database of the English language. In particular, it allows for querying nearly any word from the English language and returns the *synsets* of the word. The synset of a word w is a word that shares a similar meaning to the word w. In other words, by querying a word in WordNet, we can get the number of meanings of a word, as perceived by WordNet. Furthermore, the Pearson correlation coefficient [James et al., 2013] is computed between $\text{TPS}_n(w)$ and GS, the number of synsets for WordNet words and the word frequency as they appear in the SemEval training data, respectively. [Jakubowski et al., 2020] shows that there is a moderate (positive) correlation between $\text{TPS}_n(w)$ and GS at $n \in \{40, 50, 60\}$, a decreasing correlation between $\text{TPS}_n(w)$ and the number of synsets for WordNet words and no correlation between $\text{TPS}_n(w)$ and word frequencies.

Furthermore, we will describe how we applied topological polysemy to our word embeddings. We first implemented topological polysemy using the steps we described in Section 2.3.5, utilizing multiprocessing and the ScaNN [Guo et al., 2020] approximate nearest neighbour algorithm to speed up the computation. We chose ScaNN because it performs well when applied to word embeddings, as shown in [Aumueller et al., 2021]. We used the `ripser` [Tralie et al., 2018] Python package to compute Vietoris–Rips complexes. Next, we trained the SGNS-semeval model using the training data from the SemEval task and the hyperparameters used to train the SGNS-enwiki model from Section 3.1.3. From the training of SGNS-semeval, we got a vocabulary of size ∼122K words and a corpus of size ∼67 million. Following, we will compare the results from the experiments of [Jakubowski et al., 2020] by computing topological polysemy at varying levels of n using the word embeddings of SGNS-enwiki and SGNS-semeval. Finally, we will compare the results using the SGNS-enwiki and SGNS-semeval word embeddings to the word embeddings of the fastText.TPS.300d, GoogleNews300, glove.840B.300d and fastText.en.300d models.

We computed topological polysemy at varying levels of n using the word embeddings of the SGNS-enwiki and SGNS-semeval models, and show the results in Tables 3.7 and 3.8. In Table 3.7, we see that the correlation between TPS_n and GS is rather stable with respect to n. In particular, we notice that the correlation between TPS_n and GS is negative, suggesting a relationship in the opposite direction of the results from [Jakubowski et al., 2020, Table 1]. Nonetheless, we see a decreasing correlation when comparing TPS_n versus the number of WordNet synsets for each word, and a negligible correlation between TPS_n and word frequencies of the top 10000 most common words. Furthermore, in Table 3.8, we observe a decreasing negative correlation going towards zero between TPS_n and GS, meaning that the SGNS-semeval model performs worse than the SGNS-enwiki model on this particular task. This result indicates that by training SGNS-semeval on a smaller vocabulary than the vocabulary of SGNS-enwiki, we get worse results. Furthermore, we see a decreasing correlation between TPS_n and the number of WordNet synsets and a negligible correlation between TPS_n and word frequencies of the top 10000 most common words. Although the negative correlation between TPS_n and the number of WordNet synsets is larger for the SGNS-semeval model than the SGNS-enwiki model, it is still not particularly large. In addition to this, we are considering a lot fewer words when computing the correlation in the SGNS-semeval model than the SGNS-enwiki model (see sample size).

n	TPS_n vs. GS	TPS_n vs. synsets	TPS_n vs. frequency
10	-0.353	-0.077	**-0.043**
40	**-0.383**	-0.181	-0.041
50	-0.380	-0.190	-0.041
60	-0.381	-0.196	-0.040
100	-0.380	**-0.205**	-0.033
sample size	98	144412	10000

Table 3.7: Correlations between TPS_n and the number of word meanings as perceived by the SemEval gold standard (GS), the number of WordNet synsets and the word frequencies of the top 10000 most common words from the SGNS-enwiki model. **Bold** values indicate the largest (absolute) correlation.

n	TPS_n vs. GS	TPS_n vs. synsets	TPS_n vs. frequency
10	**-0.300**	-0.248	0.102
40	-0.201	-0.300	**0.120**
50	-0.194	-0.304	0.116
60	-0.169	-0.306	0.110
100	-0.130	**-0.310**	0.098
sample size	100	62111	10000

Table 3.8: Correlations between TPS_n and the number of word meanings as perceived by the SemEval gold standard (GS), the number of WordNet synsets and the word frequencies of the top 10000 most common words from the SGNS-semeval model. **Bold** values indicate the largest (absolute) correlation.

To further broaden our understanding of the results we got from computing topological polysemy of the word embeddings of the SGNS-enwiki and the SGNS-semeval models, we plot $\mathrm{TPS}_n(w)$ against the SemEval gold standard (GS), the number of WordNet synsets and word frequencies. We visualize the results in Figures 3.18 and 3.19, and for each plot, we let n be equal to the most optimal value for each column in Tables 3.7 and 3.8. In Figure 3.18, we see a similar situation to the results from [Jakubowski et al., 2020, Figures 8 and 9], namely that we see an indication of a linear relationship between $\mathrm{TPS}_n(w)$ and the SemEval gold standard and that we see a clear trend between $\mathrm{TPS}_n(w)$ and the number of synsets in WordNet. In Figure 3.18 (c) it is clear that there is no apparent relationship between $\mathrm{TPS}_n(w)$ and the word frequencies. Following, we see a similar situation appearing in Figure 3.19. These results suggest that, by computing $\mathrm{TPS}_n(w)$ of the SGNS-enwiki word embeddings, which has a vocabulary much larger than in the experiments of [Jakubowski et al., 2020], we are unable to use $\mathrm{TPS}_n(w)$ alone for predicting the number of word meanings, as given by the number of WordNet synsets.

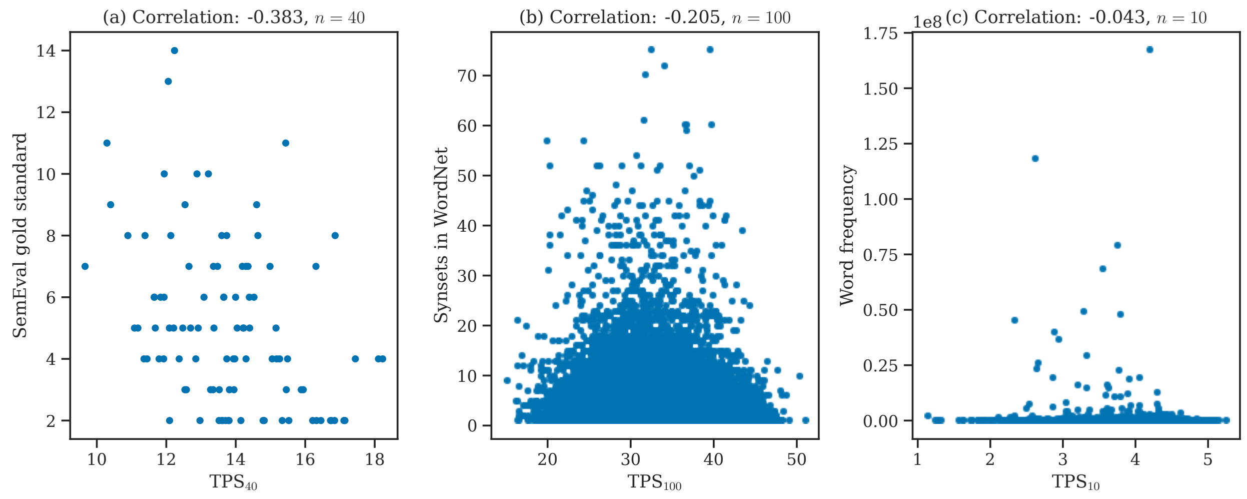

Figure 3.18: $\mathrm{TPS}_n(w)$ of the word embeddings of the SGNS-enwiki model plotted against the SemEval gold standard (GS) (a), the number of WordNet synsets (b) and word frequencies (c). The plots are inspired by [Jakubowski et al., 2020, Figures 8 and 9].

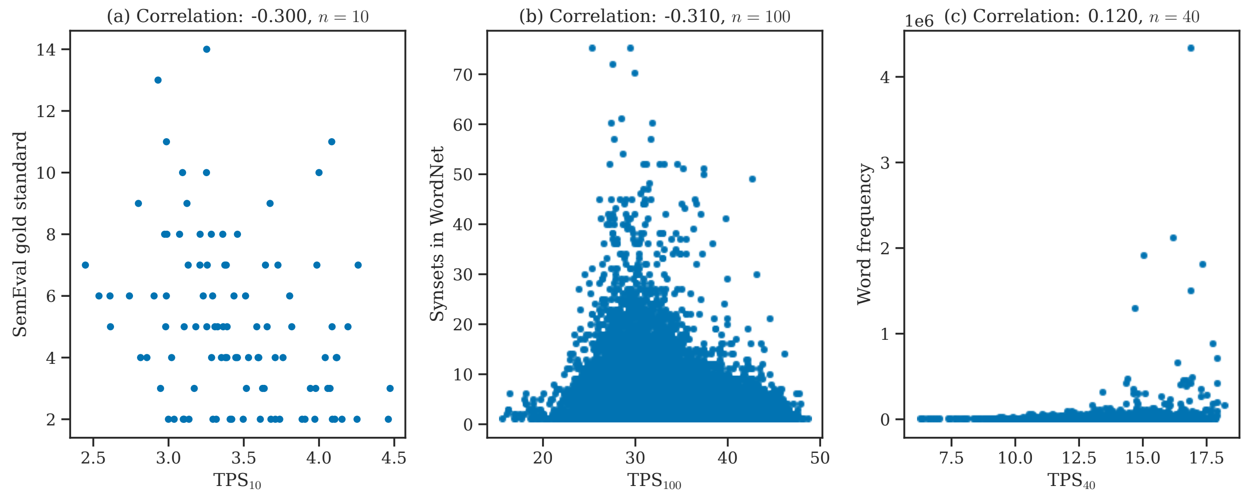

Figure 3.19: $\text{TPS}_n(w)$ of the word embeddings of the SGNS-semeval model plotted against the SemEval gold standard (GS) (a), the number of WordNet synsets (b) and word frequencies (c). The plots are inspired by [Jakubowski et al., 2020, Figures 8 and 9].

Following, we compared the results of computing $\text{TPS}_n(w)$ of the word embeddings of the SGNS-enwiki and SGNS-semeval models to the word embeddings of the fastText.TPS.300d, GoogleNews300, glove.840B.300d and fastText.en.300d models. We show the $\text{TPS}_n(w)$ results of using the fastText.TPS.300d model in Table 3.9, and using the GoogleNews300, glove.840B.300d and fastText.en.300d in Table 3.10. We did not compute the correlation between $\text{TPS}_n(w)$ and word frequencies in Tables 3.9 and 3.10, since we did not have the data available. In addition to this, it is unlikely that $\text{TPS}_n(w)$ and word frequencies have anything in common, as shown in the previous results using the SGNS-enwiki and SGNS-semeval models, as well as by the experiments of [Jakubowski et al., 2020].

n	TPS_n vs. GS	TPS_n vs. synsets
10	0.131	**0.135**
40	0.395	0.066
50	**0.416**	0.053
60	0.363	0.043
100	0.301	0.020
sample size	100	62049

Table 3.9: Correlations between TPS_n and the number of word meanings as perceived by the SemEval gold standard (GS) and the number of WordNet synsets from the fastText.TPS.300d model. **Bold** values indicate the largest (absolute) correlation.

In Table 3.9, we see similar results to the experiments of [Jakubowski et al., 2020], namely that we get a modest, positive correlation when comparing $\text{TPS}_n(w)$ to the SemEval gold standard, and that we get a decreasing correlation when comparing $\text{TPS}_n(w)$ to the number of WordNet synsets. We note, however, that we did not get exactly the same correlation results as [Jakubowski et al., 2020], which could be affected by the use of ScaNN, which approximates the nearest neighbours of words.

n	GoogleNews300		glove.840B.300d		fastText.en.300d	
	TPS_n vs. GS	TPS_n vs. synsets	TPS_n vs. GS	TPS_n vs. synsets	TPS_n vs. GS	TPS_n vs. synsets
10	**-0.446**	-0.095	-0.103	0.008	-0.240	**0.114**
40	**-0.446**	-0.166	**-0.125**	-0.039	**-0.289**	0.110
50	-0.436	-0.174	-0.053	-0.044	-0.199	0.108
60	-0.428	-0.180	-0.023	-0.048	-0.150	0.105
100	-0.417	**-0.193**	-0.053	**-0.058**	-0.105	0.099
sample size	100	207119	100	249352	100	230175

Table 3.10: Correlations between TPS_n and the number of word meanings as perceived by the SemEval gold standard (GS), and the number of WordNet synsets from the GoogleNews300, glove.840B.300d and fastText.en.300d models. **Bold** values indicate the largest (absolute) correlation.

Furthermore, in Table 3.10 we see that the GoogleNews300 model yield particularly high values when comparing $\text{TPS}_n(w)$ to the SemEval gold standard, while the remaining models are modest at best. We also observe that when comparing $\text{TPS}_n(w)$ to the number of WordNet synsets, we do not get high correlation scores. These results further suggest that by only increasing the vocabulary of the word embedding model, we are not able to model the number of WordNet synsets efficiently, only using the $\text{TPS}_n(w)$ scores. Additionally, the correlation results in Table 3.10 indicate that the $\text{TPS}_n(w)$ scores are behaving rather inconsistent across the data sets, and it is not clear if $\text{TPS}_n(w)$ measures polysemy of words.

To compare how well the various word embedding models agree on the $\text{TPS}_n(w)$, we created a correlation matrix comparing the $\text{TPS}_n(w)$ scores and the SemEval gold standard. Using a correlation matrix, we summarize the results nicely and further deepen our understanding of the results. By majority vote, we let $n = 40$ when we compared the $\text{TPS}_n(w)$ scores to the SemEval gold standard. We show the correlation matrix in Figure 3.20, where we see that the SGNS-enwiki, SGNS-semeval and GoogleNews300 models yield similar $\mathrm{TPS}_{40}(w)$ scores. We also note that the fastText.TPS.300d model either yield no correlation or negative correlations when compared to the other models.

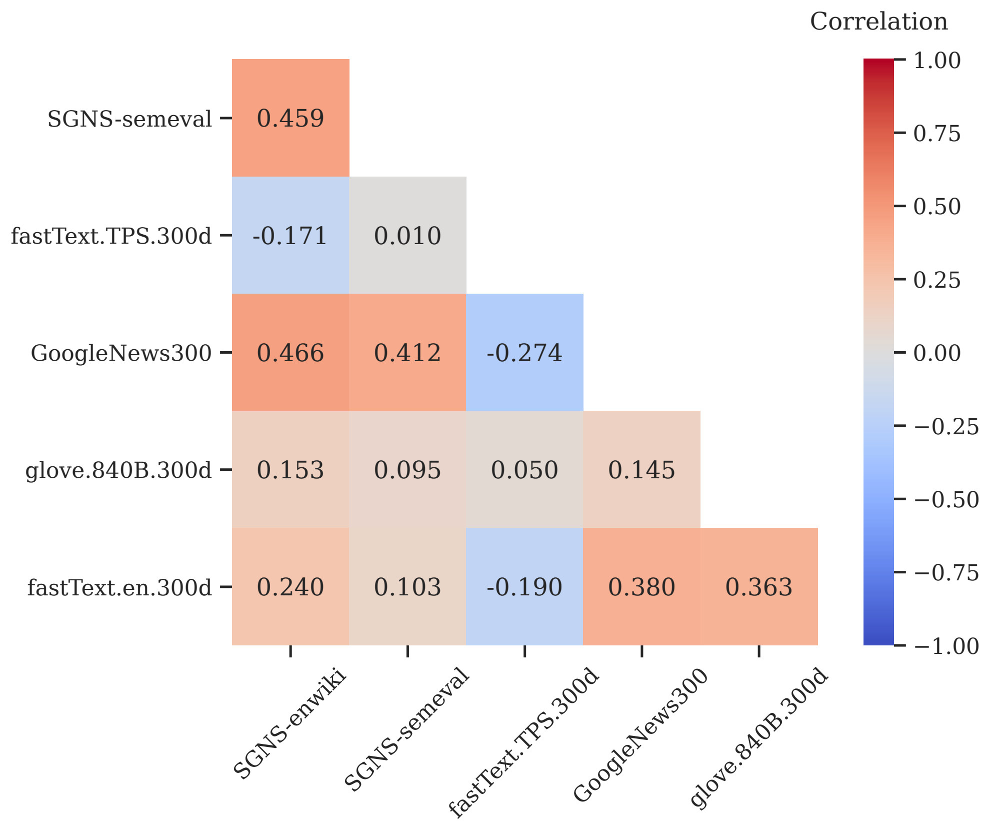

Figure 3.20: Correlation matrix comparing word embedding models on correlations between the $\mathrm{TPS}_{40}(w)$ scores and the SemEval gold standard. High (absolute) values indicate that the two models are similar in terms of scoring using $\mathrm{TPS}_{40}(w)$.

To deepen the understanding, we visualize the similarity of the SGNS-enwiki, SGNS-semeval and GoogleNews300 models in Figure 3.21, where we can see linear relationships appearing. These results suggest that the SGNS-enwiki, SGNS-semeval and GoogleNews300 models agree on how to score using $\mathrm{TPS}_{40}(w)$.

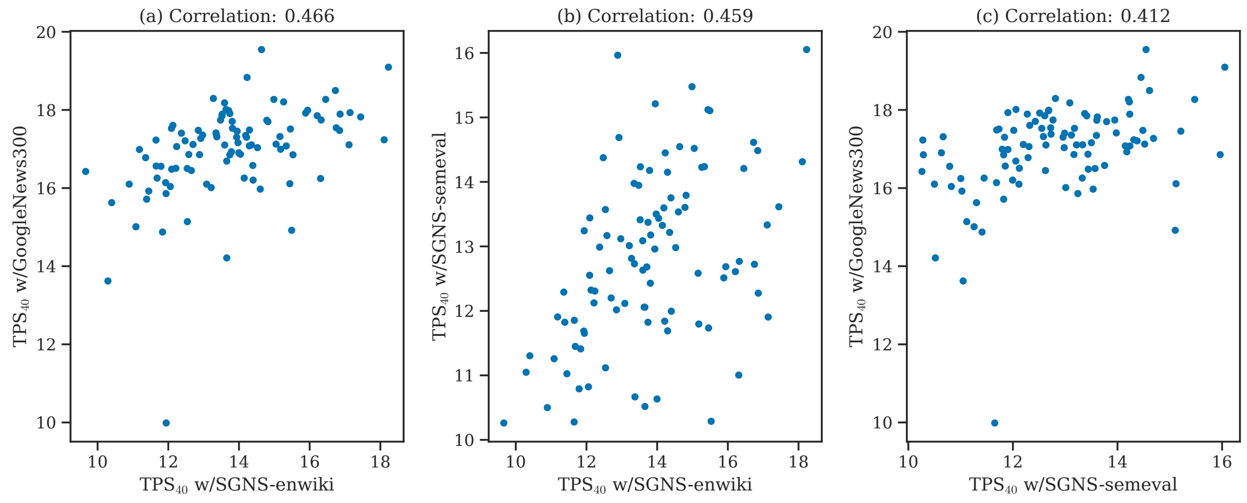

Figure 3.21: $\text{TPS}_{40}(w)$ scores plotted against each other using the SGNS-enwiki, SGNS-semeval and GoogleNews300 models.

Following, we looked at the three negative correlations which we show in Figure 3.20 and visualize the negative relationships in Figure 3.22, where we see negative relationships appearing, although it is less significant than the positive relationships seen in Figure 3.21.

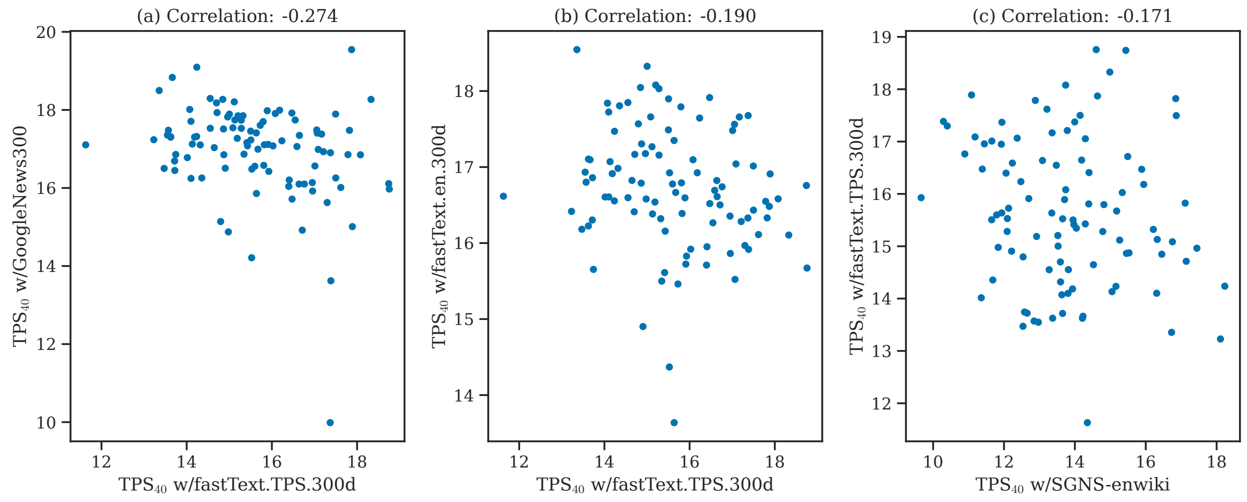

Figure 3.22: $\text{TPS}_{40}(w)$ scores plotted against each other using the fastText.TPS.300d, GoogleNews300, fastText.en.300d and SGNS-enwiki models.

We have now looked at the effect of computing $\text{TPS}_n(w)$ at varying levels of n using various word embeddings. We saw that, even by decreasing/increasing the vocabulary size of the word embedding models, the $\mathrm{TPS}_n(w)$ score did not improve significantly. In all our experiments, except using the fastText.TPS.300d model, the correlation between $\mathrm{TPS}_n(w)$ and the SemEval gold standard were always negative, while in the experiments of [Jakubowski et al., 2020], they got a moderate, positive correlation. These results suggest that the topological polysemy scoring could be affected by the choice of word embedding model, i.e. choosing fastText over word2vec, and the fact that the model used in [Jakubowski et al., 2020] was trained on a data set that is strongly related to the 100 polysemous words from the SemEval task. In other words, it could seem that the measure of topological polysemy does not work well for a general word embedding model.

To deepen our understanding of how the $\mathrm{TPS}_n(w)$ score is computed, we will perform an experiment by computing $\mathrm{TPS}_n(w)$ of a custom data set. The custom data set consists of sampled data points of two spheres that share one intersection point. We denote this data set as *2Spheres-d*, where d represents the dimensionality of the spheres. In particular, we let $d \in \{2, 3, 4, 5, 10, 20, 50, 300\}$. To ensure that the dimensionality of the *2Spheres-d* data set was similar to the dimensionality of word embeddings, we let the dimensionality of the space be equal to 300, i.e. *2Spheres-d* $\in \mathbb{R}^{300}$. In other words, if d was less than 300, we add zeros to the remaining dimensions to fill up to 300. For each sphere in *2Spheres-d*, we generate 1000000 points on the sphere in $\mathbb{R}^d$. We sort the points by distance to the intersection point and split the points into 20 intervals, i.e. chunks of 100000 data points for each sphere. Next, we sample 1000 points from each interval, leading to 20000 points for each sphere. The motivation for sampling from intervals sorted by distance was to reduce the effect of the curse of dimensionality, namely that it becomes harder to measure the distance between points in high (e.g. 300) dimension. For the sake of simplicity, we let $n = 50$ when computing the topological polysemy. We illustrate the result of computing TPS_{50} of 2Spheres-2 and 2Spheres-3 in Figure 3.23, where we see that, for both 2Spheres-2 and 2Spheres-3, the TPS_{50} is at its highest (yellow color) around the intersection point between the two spheres (see Figure 3.23 (b) and (d)). In addition to this, at the intersection point between the two spheres, the TPS_{50} score is low. These two observations suggest that for low values of d, the TPS_{50} score fails to identify the singular point and instead manages to identify the area around it. We will now look at how the TPS_{50} score behaves for $d \in \{4, 5, 10, 20, 50, 300\}$.

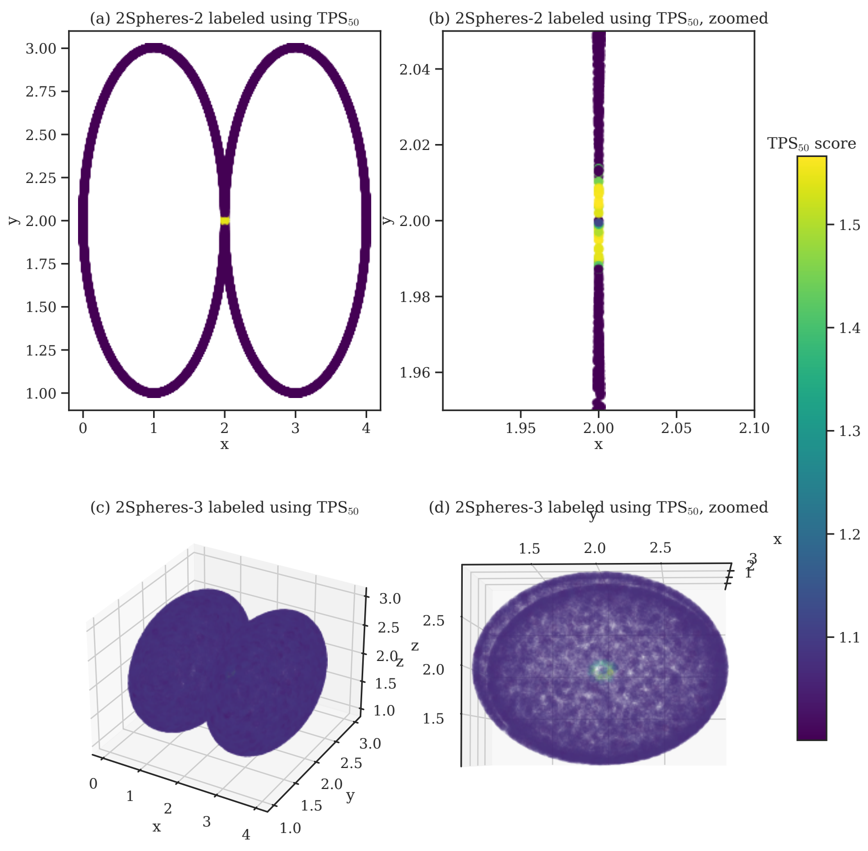

Figure 3.23: Plots of the 2Spheres-2 and 2Spheres-3 data sets, with TPS_{50} as labels. The intersection point between the spheres is at $(2, 2)$ in 2 dimensions and at $(2, 2, 2)$ in 3 dimensions.

We visualize the result of computing TPS_{50} of 2Spheres-d for $d \in \{4, 5, 10, 20, 50, 300\}$ in Figure 3.24, by plotting the distance to the intersection point between the spheres against the TPS_{50} scores. In Figure 3.24, we see that as the dimension of the spheres increases, the ”peak” of TPS_{50} scores close to the intersection point diminishes. The diminishing effect comes due to the curse of dimensionality, namely that in high dimensional space, all distances become similar, as seen in Figure 3.24 (f). In other words, for high dimensional spheres, it becomes difficult to identify the intersection point between the spheres, using the TPS_{50} scores, as the distances become similar, and TPS_{50} is unable to identify areas around the intersection point, which we saw happening in lower dimensions in Figure 3.23. We note, however, however, that for high values of d, the intersection point has a TPS_{50} score which generally is higher than all other values of TPS_{50}. Finally, we argue that the results in Figure 3.24 indicate that the topological measure of polysemy may suffer when applied to high-dimensional (e.g. 300) data.

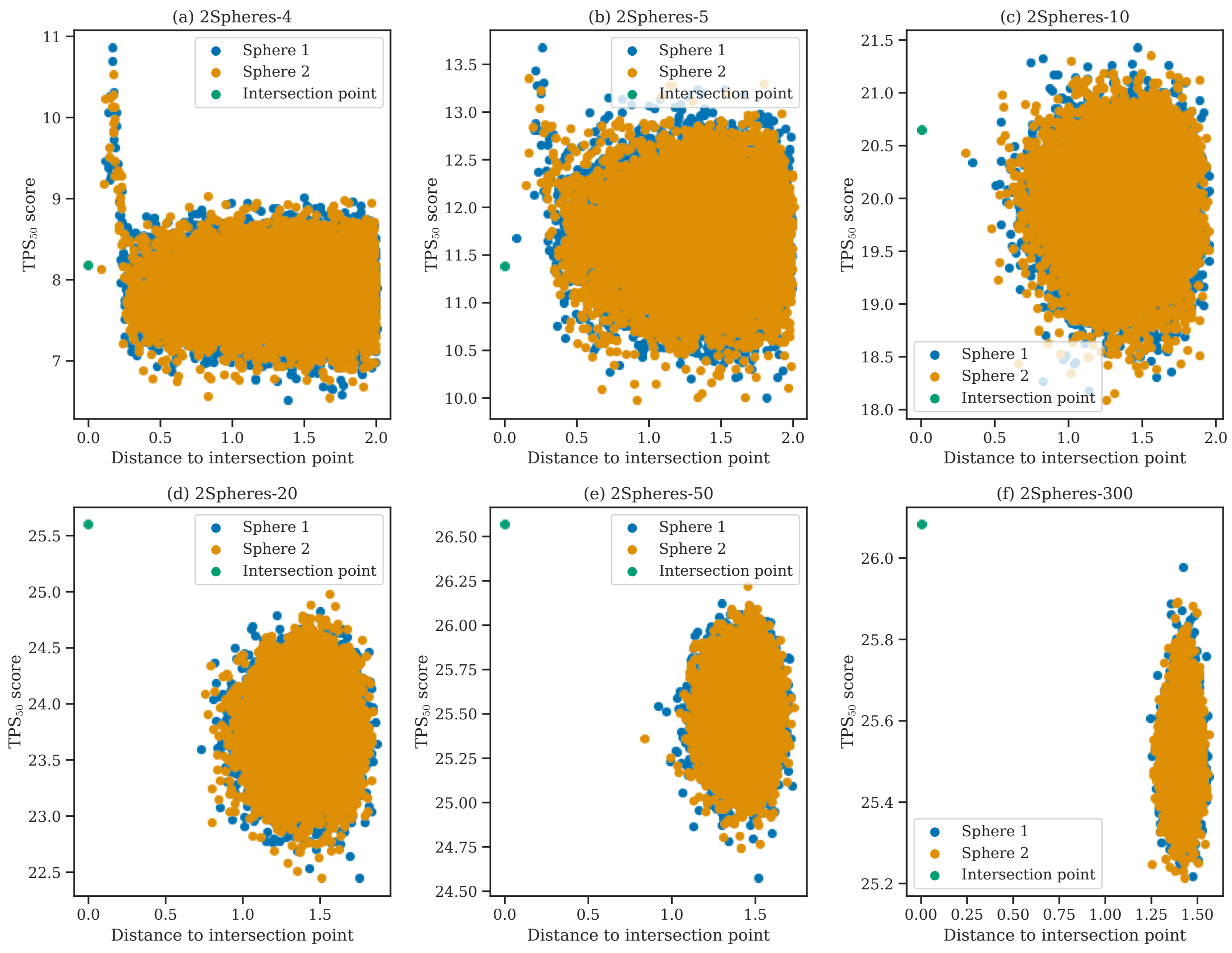

Figure 3.24: Distance to the intersection point between spheres plotted against TPS_{50} scores for 2Spheres-d, $d \in \{4, 5, 10, 20, 50, 300\}$.

Following, we repeated the experiment where we computed the topological polysemy of two spheres. In particular, we used a noisy version of the 2Spheres-d data set, which we denoted as the *2SpheresNoisy-d* data set. The motivation for adding some noise to the spheres data set was to emulate some real-world effect since real-world data sets are usually not uniformly distributed. In particular, the 2SpheresNoisy-d data set was created by perturbing the 2Spheres-d data set by adding Gaussian noise at every data point. In particular, we used Gaussians with zero mean and a variance of 0.1. We first computed the TPS_{50} score of the 2SpheresNoisy-2 and 2SpheresNoisy-3 data sets. We show the results in Figure 3.25, where we see that in the 2-dimensional case, the TPS_{50} scores are high (i.e. yellow colour) and we are unable to identify the intersection point between the spheres. Moreover, in Figure 3.25 (c) and (d) we see that the TPS$_{50}$ scores are mediocre at best, and we are unable to identify the intersection point between the spheres. The results in Figure 3.25 tells us that if we perturb the data set by adding noise, it becomes harder to identify singular points in lower dimensions ($d \in \{2, 3\}$), when compared to the TPS$_{50}$ results using 2Spheres-d, which we show in Figure 3.23.

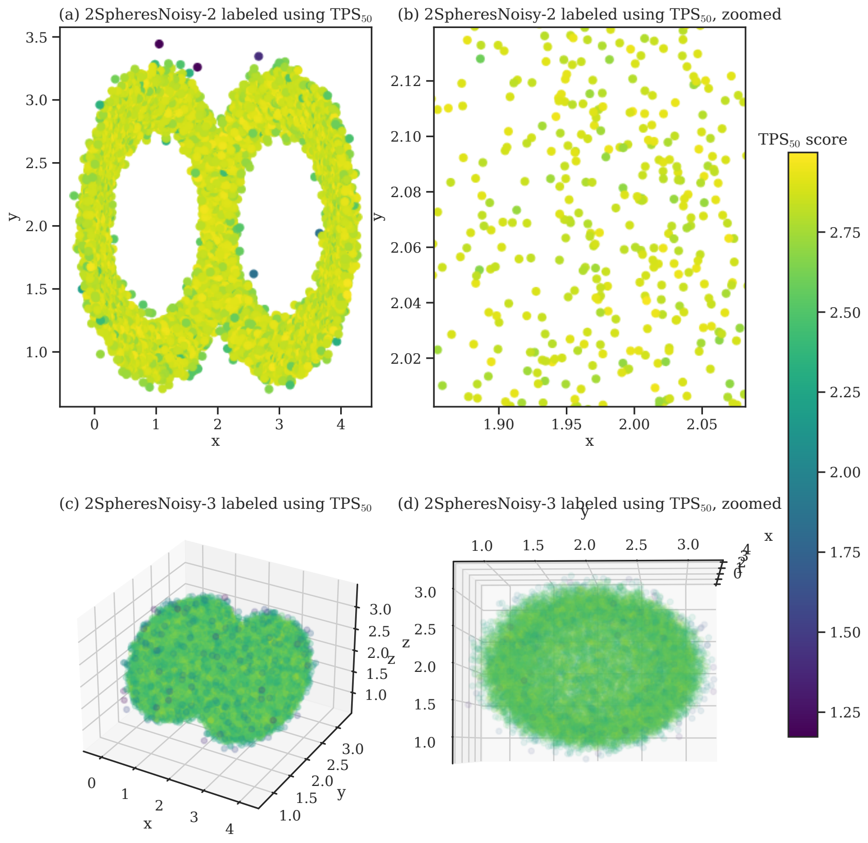

Figure 3.25: Plots of the 2SpheresNoisy-2 and 2SpheresNoisy-3 data sets, with TPS$_{50}$ as labels. The intersection point between the spheres is at $(2, 2)$ in 2 dimensions and at $(2, 2, 2)$ in 3 dimensions.

Finally, we visualize the result of computing TPS$_{50}$ for 2SpheresNoisy-d for $d \in \{4, 5, 10, 20, 50, 300\}$ in Figure 3.26, by plotting the distance to the intersection point between the spheres against the TPS$_{50}$ scores. In Figure 3.26, we see a similar situation appearing to the results using 2Spheres-d in Figure 3.24. In particular, we observe that as we increase the dimensionality of the spheres towards 300, the distances to the intersection point becomes more or less the same, and it is difficult to differentiate between the intersection point and regular points on the two spheres. We note, however, that in Figure 3.26 (f) we see that the TPS_{50} score is significantly larger than the rest of the TPS_{50} scores and it could be possible to identify the intersection point using its TPS_{50} score. Although this result might seem significant in Figure 3.26 (f), the 2SpheresNoisy-d is still rather simple when compared to real-world word embeddings, even when we added the noise. Additionally, the significance we show in Figure 3.26 (f) can be due to a random effect of the 2SpheresNoisy-300 data set.

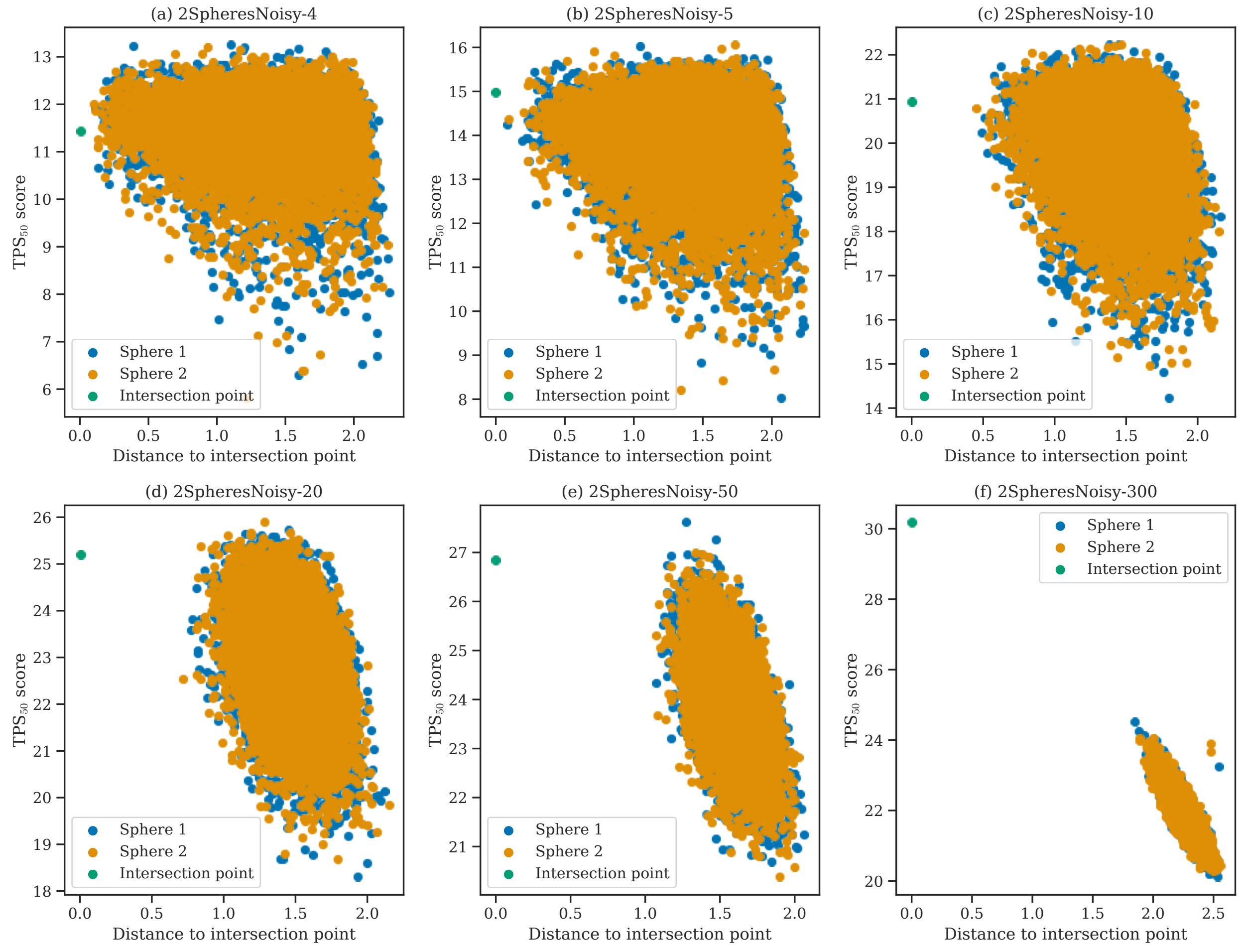

Figure 3.26: Distance to the intersection point between spheres plotted against TPS_{50} scores for 2SpheresNoisy-d, $d \in \{4, 5, 10, 20, 50, 300\}$.

Furthermore, we will use the measure of topological polysemy when we create supervised models for polysemy prediction in Section 3.3.4. Next, we will look at Geometric Anomaly Detection, and in particular, how it performs when applied to word embeddings.

3.3.2 Geometric Anomaly Detection

In this subsection, we will apply the Geometric Anomaly Detection (GAD) algorithm (see Section 2.3.6) to the word embeddings from the SGNS-enwiki model. In particular, we will show the relationship between how GAD categorizes word embeddings into groups and whether words are polysemous. We implemented GAD as explained in Section 2.3.6, using similar packages to the ones we used to implement topological polysemy in Section 3.3.1. In particular, we used the ScaNN [Guo et al., 2020] approximate nearest neighbour algorithm, to speed up the nearest-neighbour computation, and `ripser` [Tralie et al., 2018] Python package, to compute Vietoris–Rips complexes. We also included an option to use the Ripser++ [Simon Zhang and Wang, 2020] Python package instead of `ripser`, which is a GPU accelerated version. However, we quickly found that the GPU overhead was too big and it was faster just to use the regular `ripser` Python package.

Before applying GAD to word embeddings, we will motivate the use of GAD by visualizing GAD applied to the 3-dimensional *Henneberg surface* data set, as used in the experiments of [Stolz et al., 2020]. To compute the GAD of the Henneberg surface data set, we used the same hyperparameters as in [Stolz et al., 2020], i.e. we let the inner annulus radius equal 1.5, outer annulus radius equal 2 and the manifold dimension k equal 2. We visualize the result in Figure 3.27, where we see how GAD groups data points to the manifold, boundary and singular groups. In the 3-dimensional Henneberg surface data set, four 2-dimensional surfaces intersect, and as we see in Figure 3.27 (b), where the GAD algorithm managed to identify the intersecting points as singular points. In addition to the singular points, we see the boundary points quite clearly in Figure 3.27 (a) and (b).

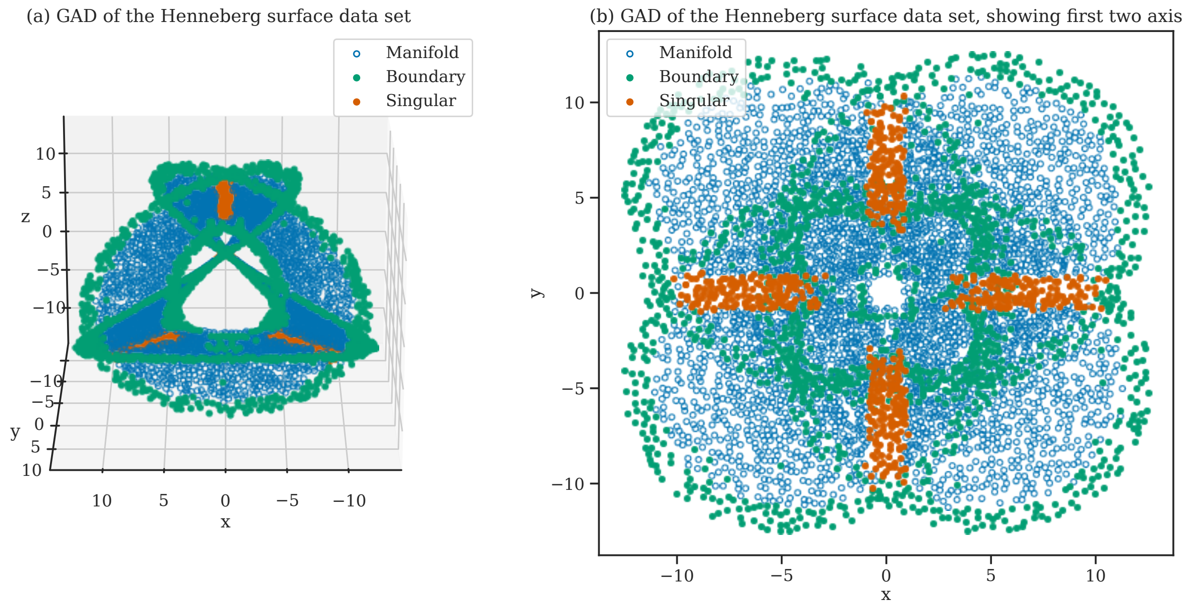

Figure 3.27: 2D and 3D projections of the Henneberg surface data set. The data points are labelled using their groups from the GAD algorithm. This figure is inspired by [Stolz et al., 2020, Figure 3].

Following, we visualize the Henneberg surface data set with the TPS_{50} score computed for each point. In Figure 3.28, we see that the TPS_{50} scores fails to identify the singular data points if the Henneberg surface data set,which we expected to have relatively high TPS_{50} scores. In particular, the TPS_{50} scores are relatively high for points on the manifold, and lower for the boundary and singular points.

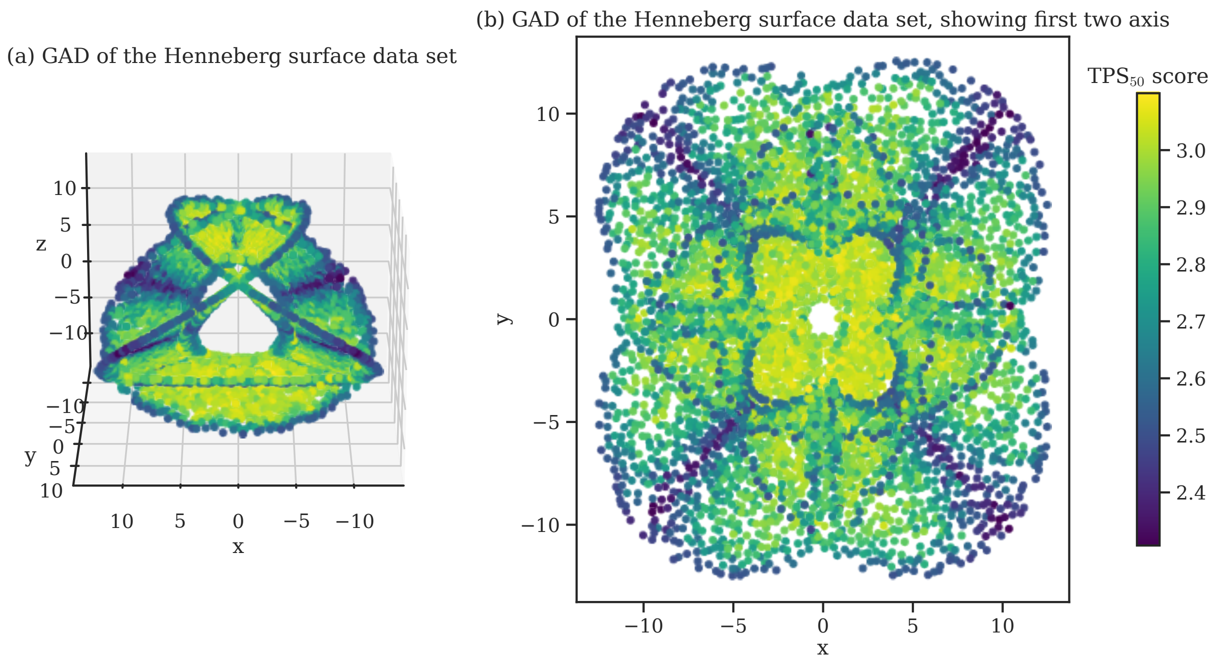

Figure 3.28: 2D and 3D projections of the Henneberg surface data set. The data points are labelled using their TPS$_{50}$ scores. This figure is inspired by [Stolz et al., 2020, Figure 3].

We will now apply GAD to the word embeddings from the SGNS-enwiki model. In particular, we will apply GAD to the words that have a WordNet entry, similar to how we performed the topological polysemy experiments on SGNS-enwiki word embeddings in Section 3.3.1. Furthermore, the standard GAD algorithm uses annulus radii parameters to do its computation, and because it is unknown which radii parameters to use for a general data set, we will instead default to a k-nearest neighbour approach when computing GAD of the WordNet SGNS-enwiki word embeddings. Using the k-nearest neighbour approach, we let the inner annulus radius equal the distance to the s nearest neighbour of each word, and similarly for the outer annulus radius, which we set equal to the distance to the t nearest neighbour of each word. To apply GAD to the WordNet SGNS-enwiki word embeddings, we used the parameters $s = 25$ and $t = 500$. We show the number of words in each GAD group in Table 3.11, where we see that the number of polysemous WordNet words that fall into the singular group is particularly low, i.e. only 344 of 48880. In addition to this, we see that the number of words falling into the boundary group words is high. These two observations suggest that our inner and outer annulus radius, as well as the manifold dimension k, were not set correctly for the data set we are applying GAD to. Keep in mind that the intrinsic dimensionalities of word embeddings are most likely higher than $k = 2$, but due to the computational cost of setting $k > 2$ (creation of the Vietoris–Rips complex), we will not set k greater than 2 in this thesis.

	GAD group			
	Manifold	Boundary	Singular	*Sum*
Number of monosemous words	4640	86731	4161	95532
Number of polysemous words	634	47902	344	48880
Sum	5274	134633	4505	144412

Table 3.11: Comparison of the number of monosemous and polysemous words that belong to the various GAD groups, when we compute GAD of the WordNet SGNS-enwiki word embeddings.

Finally, we visualize the result using a 2-dimensional UMAP embedding of the 10000 most common words of the WordNet SGNS-enwiki word embeddings in Figure 3.29, which we labelled using the GAD groups. We created the UMAP embedding using the default hyperparameterization of the `umap-learn` Python package. In Figure 3.29, we see that only a single word is categorized to be singular (the word ”branch”), some words are categorized to be on the manifold, and the rest are categorized to be on the boundary. These results indicate that the hyperparameters we used to compute GAD are not suitable for the data set at hand. We also observe that Figure 3.29 differs a whole lot from GAD applied to the Henneberg surface data set (see Figure 3.27), which could indicate bad hyperparameterization.

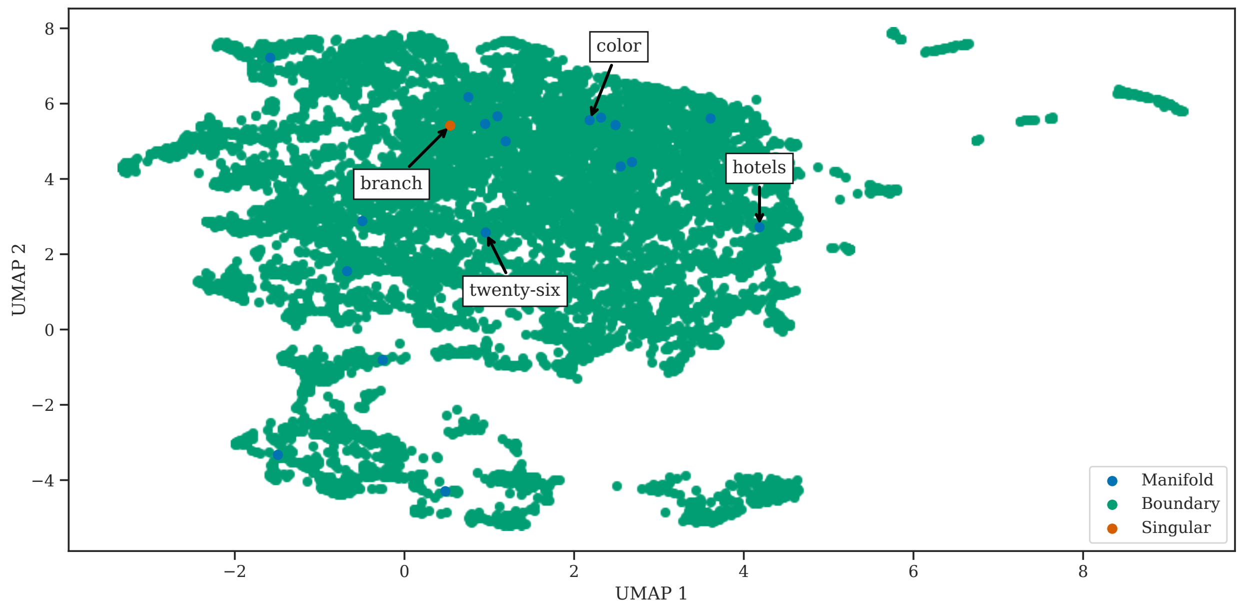

Figure 3.29: 2-dimensional UMAP embedding of the 10000 most common words of the WordNet SGNS-enwiki word embeddings. The points are labelled using their respective GAD groups.

Furthermore, we will investigate the effect of using different sets of hyperparameters when applying GAD to word embeddings in Section 3.3.4, where we create supervised models for polysemy prediction. Next, we will investigate algorithms for estimating the intrinsic dimensionality of word embeddings, and show how it correlates with the actual number of word meanings.

3.3.3 Intrinsic dimension estimation

In this subsection, we will look at intrinsic dimension (ID) estimation algorithms (see Section 2.1.6) and apply them to word embeddings. In particular, we will apply ID estimation algorithms to the WordNet SGNS-enwiki word embeddings, used in experiments in Section 3.3.1 and Section 3.3.2. We will show the relationship between the estimated ID and the number of WordNet word meanings. To demonstrate the relationship between estimated ID and number of WordNet word meanings, we will use the LPCA (Section 2.1.6.1), TWO-NN (Section 2.1.6.3) and TLE (Section 2.1.6.5) algorithms. For each of the ID estimation algorithms, we used the 200 nearest neighbours of each word to estimate their local IDs, using the `scikit-dimension` Python package [Bac, 2020]. Following, we plot the estimated IDs versus the number of WordNet word meanings in Figure 3.30, where we observe a similar situation appearing to the ones we see in Figure 3.18 and Figure 3.19. Particularly, we see a clear trend when plotting the estimated IDs to the number of WordNet word meanings. We also see that the different ID estimation algorithms yield different results: LPCA estimates ID up to 120, while TWO-NN and TLE estimate ID up to 50 and 60. These results suggest that we can not simply rely on a single estimate of the ID, and it could be useful to use multiple ID estimates since they are measured differently (see Section 2.1.6 for more details).

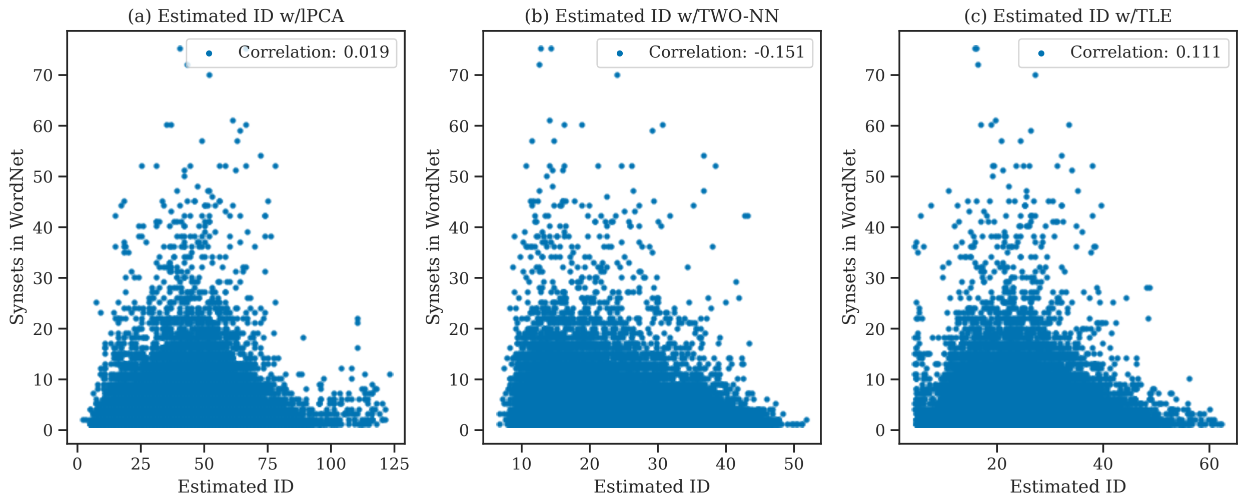

Figure 3.30: Estimated IDs plotted against the number of word meanings, using LPCA, TWO-NN and TLE ID estimation algorithms.

We have now shown the relationship between estimated IDs and the number of word meanings. In the next section, we will create supervised models for estimating the number of word meanings and predicting whether or not a word is polysemous. We will use multiple sets of hyperparameters and all ID estimation algorithms specified in Section 2.1.6, as well the topological polysemy (Section 3.3.1) and Geometric Anomaly Detection (Section 3.3.2) algorithms.

3.3.4 Supervised polysemy prediction

In this subsection, we will propose two supervised models to predict the number of word meanings. As we have seen in the previous subsections (Section 3.3.1, Section 3.3.2 and Section 3.3.3), the number of word meanings seem to be more or less correlated with topological polysemy, Geometric Anomaly Detection (GAD) and intrinsic dimension (ID) estimation. For this reason, we will propose two supervised model using lasso regression (Section 2.1.7.2) and logistic regression (Section 2.1.7.3), incorporating the results from topological polysemy, GAD and ID estimation. The goal of these models is to use the results from the topological polysemy, GAD and ID estimation methods to improve on the prediction of the number of word meanings. We chose to use lasso regression because it has feature importance packed in the model. The feature importance part is important for us because we would like to try multiple configurations of hyperparameters for each algorithm used to create the training data. The logistic regression model is trained using ℓ_1-penalty, allowing the model to perform feature importance. The lasso regression model tries to predict the number of word meanings, while the logistic regression model performs binary prediction of whether or not a word is polysemous. We also attempted to create a multi-class (e.g. one meaning, two meanings, etc.) model using multinomial logistic regression, but it became apparent that the problem was too hard and we decided not to follow up with those experiments. Furthermore, we denote the lasso regression model as *WME-enwiki* (short for **W**ord **M**eaning **E**stimation-enwiki) and the logistic regression model as *BWME-enwiki* (short for **B**inary **W**ord **M**eaning **E**stimation-enwiki). Next, we will describe the creation of training data used for both supervised models, before going into detail about the training and evaluation process.

To create the training data used in the WME- and BWME-enwiki models, we used the word embeddings from the SGNS-enwiki model. In particular, we used the word embeddings that have a WordNet entry, resulting in 144412 words. We denote these word embeddings as the WordNet SGNS-enwiki word embeddings. The number of word meanings (i.e. the number of WordNet synsets) is used as labels y for the WME-enwiki model. For the BWME-enwiki model, we used binary labels, i.e. $y = 0$ if the word had exactly one word meaning, and $y = 1$ if the word had two or more meanings.

To create the features of the training data, we first computed topological polysemy $\text{TPS}_n(w)$ of the WordNet SGNS-enwiki word embeddings. We computed $\text{TPS}_n(w)$ at for varying $n = 10, 20, 30, \ldots, 250$ (step size of 10, leading to 25 values of n) and used them as features in the data. In addition to this, we computed the maximum, average and standard deviation of the birth values of the zero-degree persistence diagram computed by $\text{TPS}_n(w)$, leading to 3 additional features for each $\text{TPS}_n(w)$. In total, this resulted in 25 (values of n) $\times\ 4 = 100$ features from topological polysemy.

Following, we applied GAD to the WordNet SGNS-enwiki word embeddings. To compute GAD, we used the k-nearest neighbour version, similar to the experiments of Section 3.3.2; we let the inner annulus radius equal the distance to the s-nearest neighbour and the outer annulus radius equal the distance to the t-nearest neighbour. Since we used the k-nearest neighbour version of GAD we were more in control of the computation time, because setting the radius manually can lead to large and difficult computations of the Vietoris–Rips complex, as some areas are denser than others. We show the different choices of s and t in Table 3.12, which leads to 23 different configurations of the inner and outer annulus k-nearest neighbours. We let the manifold dimension k equal 2 for all words, even though the local intrinsic dimension for each word is likely higher than 2. This was done to make the GAD computation feasible within the computational resources at hand; we will revisit the manifold dimension choice when discussing future work in Chapter 5. For each of the (s, t) configurations used to parameterize GAD, we created one feature for each GAD group (i.e. manifold, boundary and singular) as 3-dimensional one-hot encodings. For example, if a word is categorized as being on the manifold, then its value is equal to 1 and the rest are set to zero. In other words, we are left with 23 (configurations) $\times$ 3 (GAD groups) = 69 features from GAD. We also attempted to vectorize the persistence diagrams created by GAD using persistence images (Section 2.3.3), but it quickly led to far too many features as we used each pixel in the images as a separate feature, and we were unable to train the WME- and BWME-enwiki models efficiently. We will revisit the use of persistence images when discussing future work in Chapter 5.

Inner annulus, s-nearest neighbour	Outer annulus, t-nearest neighbour
25	250
25	500
25	750
25	1000
50	250
50	500
50	750
50	1000
100	1000
100	1250
100	1500
100	1750
100	2000
150	1000
150	1250
150	1500
150	1750
150	2000
200	1000
200	1250
200	1500
200	1750
200	2000

Table 3.12: Our configurations of s, i.e. inner annulus nearest neighbour, and t, i.e. outer annulus nearest neighbour, for computing GAD of the WordNet SGNS-enwiki word embeddings.

Furthermore, we estimated the local ID of the WordNet SGNS-enwiki word embeddings using the ID estimation algorithms from Section 2.1.6. More precisely, we used the LPCA (Section 2.1.6.1), KNN (Section 2.1.6.2), TWO-NN (Section 2.1.6.3), MLE (Section 2.1.6.4) and TLE (Section 2.1.6.5) algorithms. For each of the ID estimation algorithms, we used the k-nearest neighbours of each word to estimate their local IDs. We used the following values for k: 25, 50, 100, 150 and 200. The estimated local ID of each word is used as a feature in the training data, leading to 5 (algorithms) $\times$ 5 (values for k) $= 25$ features from ID estimation. We used the `scikit-dimension` Python package [Bac, 2020] to estimate the local IDs.

In total, the training data had 100 (from topological polysemy) + 69 (from GAD) + 25 (from ID estimation) = 194 features. Following, we split the training data into three new distinct data sets (as motivated by Section 2.1.8.1): training, test and SemEval test data sets. The new training data set consisted of 95% random words of the original training data set, excluding the 100 SemEval-2010 Task 14 target words (as used in Section 3.3.1). The test data set consisted of 5% random words of the original training data set, excluding the 100 SemEval-2010 Task 14 target words. We will use the test data to evaluate the performance of the trained WME- and BWME-enwiki models. Furthermore, the SemEval test data set consisted of the 100 SemEval-2010 Task 14 target words and will be used to evaluate the performance using the WME-enwiki model. We emphasize that the training, test and SemEval test data sets do not have overlapping words, as we do not want to be training on words from the test data sets. The training data set consisted of 137098 words, the test data set consisted of 7216 words, and the SemEval test data set consisted of 98 words (as 2 of the words were out of the SGNS-enwiki vocabulary). For each data set, we transformed the features by removing the mean and scaling to unit variance, as we did not want the WME- and BWME-enwiki models to be affected by different means and variances across the features. For the SemEval test data set, we used the SemEval gold standard as the number of word meanings, while for the training and test data set we used the number of WordNet synsets as the number of word meanings.

Following, we trained the WME- and BWME-enwiki models using k-fold cross-validation (Section 2.1.8.2). We found $k = 20$ to work well with our data, meaning that we used 6855 random words for each fold in the cross-validation. For the WME-enwiki model, we cross-validated over 10000 values of λ, starting from $\lambda = 0.0000001$ to $\lambda = 0.01$. We found the most optimal value of λ for the WME-enwiki model to be 0.0000291. For the BWME-enwiki model, we cross-validated over 10000 values of λ, starting from $\lambda = 0.00001$ to $\lambda = 0.01$. We found the most optimal value of λ for the BWME-enwiki model to be 0.000692. To perform the cross-validation we used the `LassoCV` and `LogisticRegressionCV` classes from `scikit-learn` for the WME- and BWME-enwiki models, respectively. For the WME-enwiki model, we used the default scoring of the `LassoCV` class. On the other hand, for the BWME-enwiki model, our goal was to maximize the ability of the model to predict polysemous words accurately. As such, we used the sensitivity metric (Section 2.1.9.3) to score the folds from the BWME-enwiki cross-validation. Furthermore, we show the results from training the WME-enwiki model in Figure 3.31, where we see a weak correlation between the predicted number of word meanings and the number of WordNet synsets for both the training and test data sets. In Figure 3.31 (c), we see that the model is unable to predict the number of word meanings for the SemEval data set, which is not surprising, as we have trained using the number of WordNet synsets. We note, however, that we see clear trends in all plots of Figure 3.31.

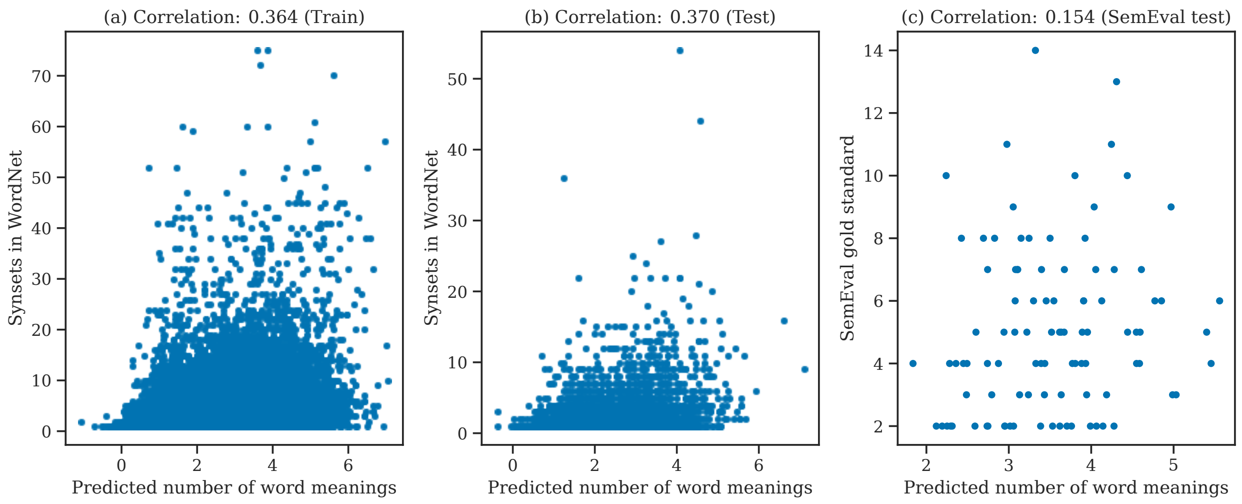

Figure 3.31: The predicted number of word meanings plotted against the number of WordNet synsets and SemEval gold standard, using the WME-enwiki model.

By looking at the values of the feature coefficients of the WME-enwiki model, we saw how the model prioritized certain features over others. We show the top 10 most important features in Figure 3.32, where we observe that the features from topological polysemy, for large values of n, are the most relevant for the model. The MLE and TLE intrinsic dimension estimators are also relatively relevant for high values of the k-nearest neighbour. The features from GAD are not in the top 10 most important features.

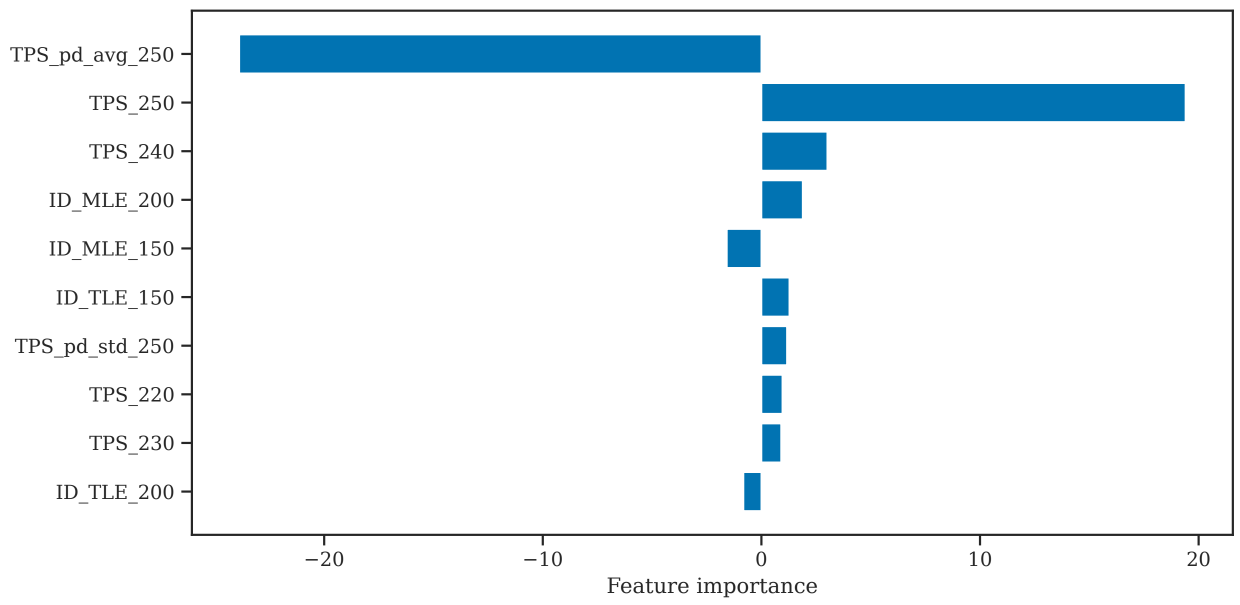

Figure 3.32: Feature importances (i.e. coefficients) for the top 10 most important features of the WME-enwiki model. We sort the feature importances by their absolute values in descending order.

To investigate the feature importances for the TPS, GAD and ID estimator features, we visualize the top 10 most important features in Figure 3.33, where we see that the TPS_{250} features are especially relevant. From the GAD features in Figure 3.33 (b), we see that whether or not a word is classified as boundary or singular is important, while being classified as a manifold is not as relevant. Finally, we see that the MLE and TLE ID estimator methods yield important features for various values of n. We note, however, that the features importances in Figure 3.33 (a) and (c) are more important than the feature importances in Figure 3.33 (b), as noted by the x-axis scales.

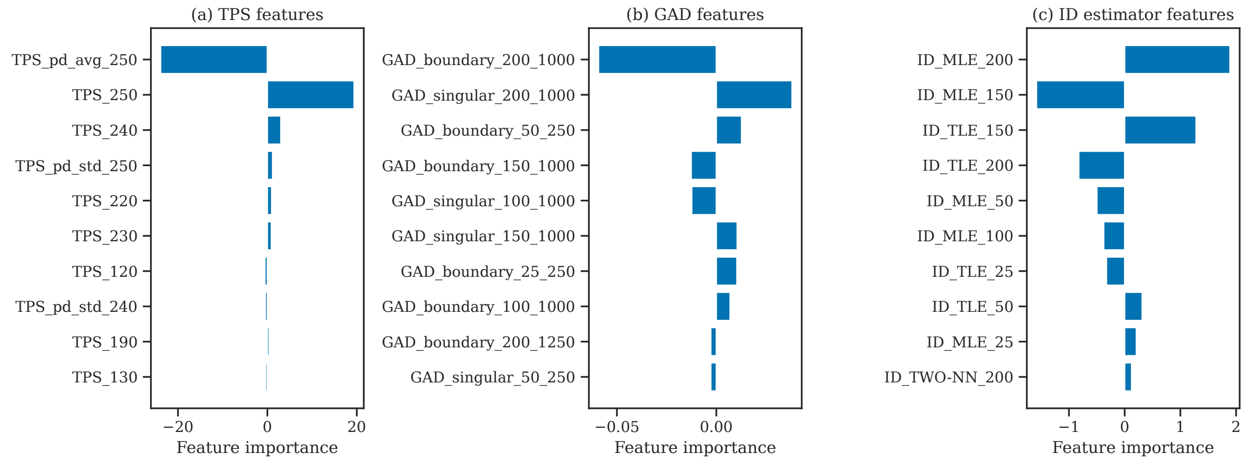

Figure 3.33: Feature importances (i.e. coefficients) of the top 10 TPS, GAD and ID estimator features, using the WME-enwiki model. We sort the feature importances by their absolute values in descending order.

From the training of the WME-enwiki model, the lasso set some of the features to zero, essentially removing them from the model. In particular, 48 of 194 features were set to zero, and most of them were various configurations of GAD which did not yield any interesting result (e.g. all words classified as boundary words). We have now looked at the results from training the WME-enwiki model, and in particular, looked at its performance for predicting the number of word meanings and which features were important to the model.

Next, we will look at the results from the training of the BWME-enwiki model. We show the results from the training of the BWME-enwiki model in Figure 3.34, where we see the result of predicting the number of word meanings on the training and test data sets using confusion matrices (Section 2.1.9.2). As we show in Figure 3.34, we get a sensitivity of 0.393 on the train data sets, meaning that the model identifies 39.3% of the polysemous words. The test sensitivity shows that the model identifies 39.4% of all the unseen polysemous words. These results indicate that the model can not efficiently predict whether or not a word is polysemous, as we ideally would like the sensitivity on both the training and test sets to be at least 0.5 (or 50%).

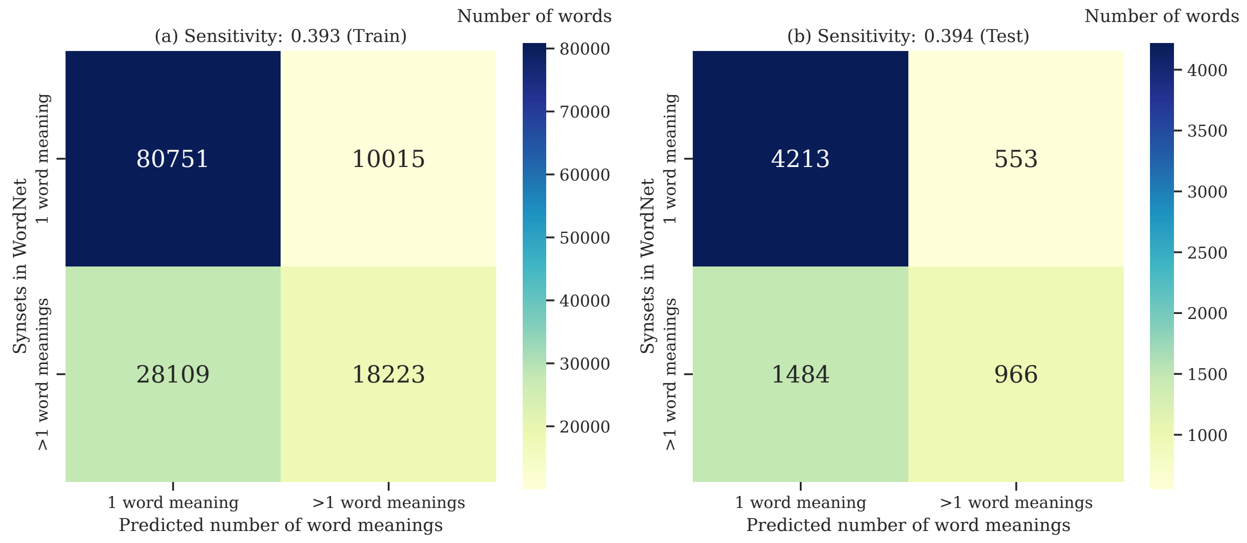

Figure 3.34: Confusion matrices for predicting the number of word meanings, using the BWME-enwiki model. The first confusion matrix (a) shows the result on the training data set, while the second confusion matrix (b) shows the result on the test data set.

To deepen our understanding of which words the BWME-enwiki model had a harder time with, we looked at the misclassified monosemous and polysemous test words. Of the 1484 words the BWME-enwiki model incorrectly predicted to be monosemous, we report the top 10 most common misclassified test words, namely the following words: ”time”, ”age”, ”returned”, ”Italian”, ”Chicago”, ”gold”, ”tower”, ”jones”, ”unable” and ”opposition”. Furthermore, of the 553 words the BWME-enwiki model incorrectly predicted to be polysemous, we report the top 10 most common misclassified test words, namely the following words: ”january”, ”ninety-six”, ”ninety-one”, ”seventy-one”, ”sixty-three”, ”fifty-four”, ”fifty-eight”, ”non”, ”citizens” and ”additionally”. From these sets of words, we do not see any particular pattern. Furthermore, we visualize the correctly and incorrectly classified words from the test data set in Figure 3.35, using a 2-dimensional UMAP embedding. In Figure 3.35 (b) and (c), we emphasize the misclassified words, and we do not see any particular pattern here either, as the words are spread throughout the UMAP embedding.

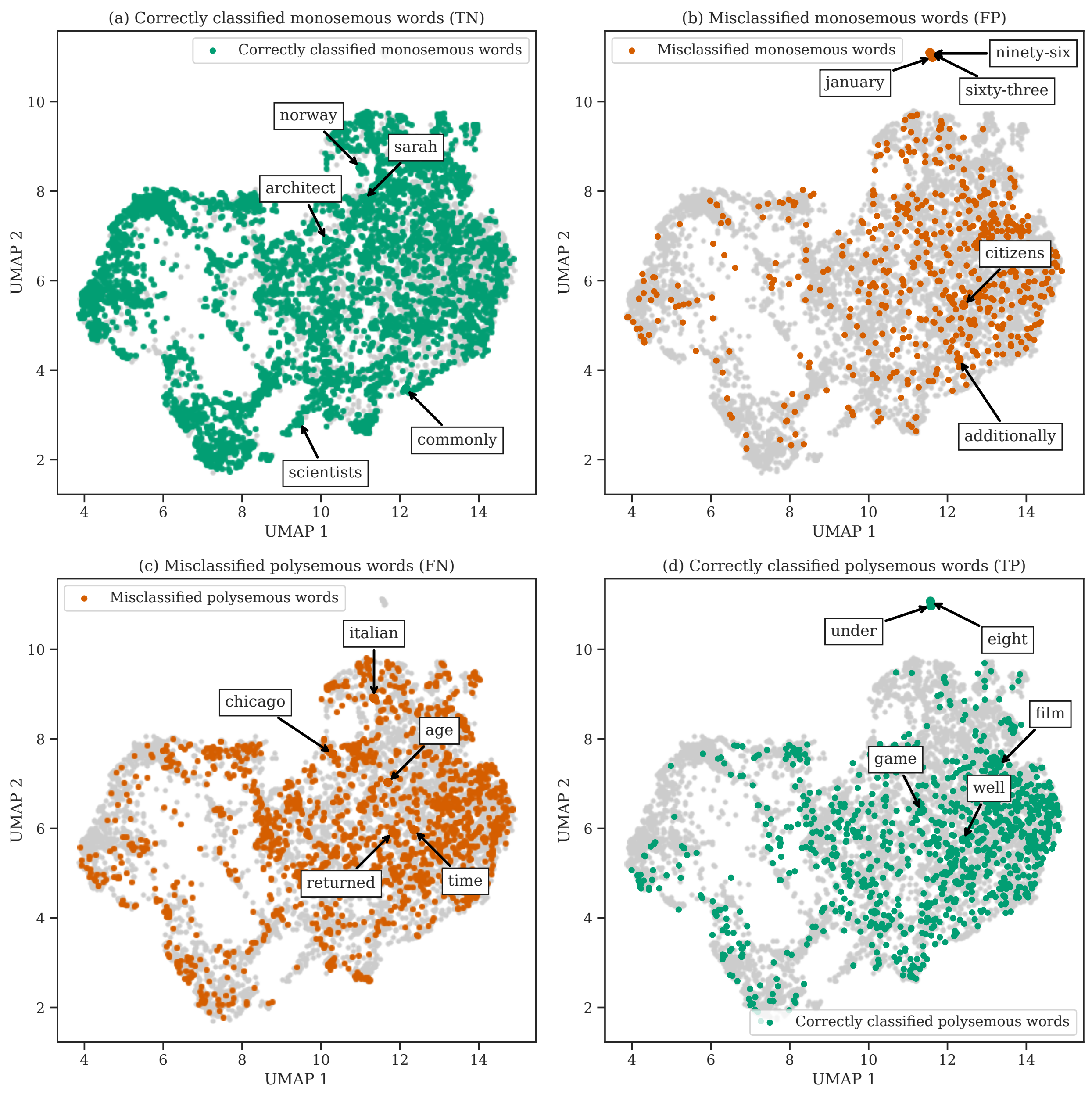

Figure 3.35: 2-dimensional UMAP embedding of the test data set evaluated on the BWME-enwiki model. We emphasize the correctly and incorrectly classified words in a "correlation matrix" fashion.

Next, we will investigate the feature importance in the BWME-enwiki model by looking at the coefficient values of the features. We show the top 10 most important features of the BWME-enwiki model in Figure 3.36, where we see a similar pattern to the top 10 feature importances of the WME-enwiki model, i.e. that the TPS features (for varying n) are most important, followed by the features from the ID estimator models.

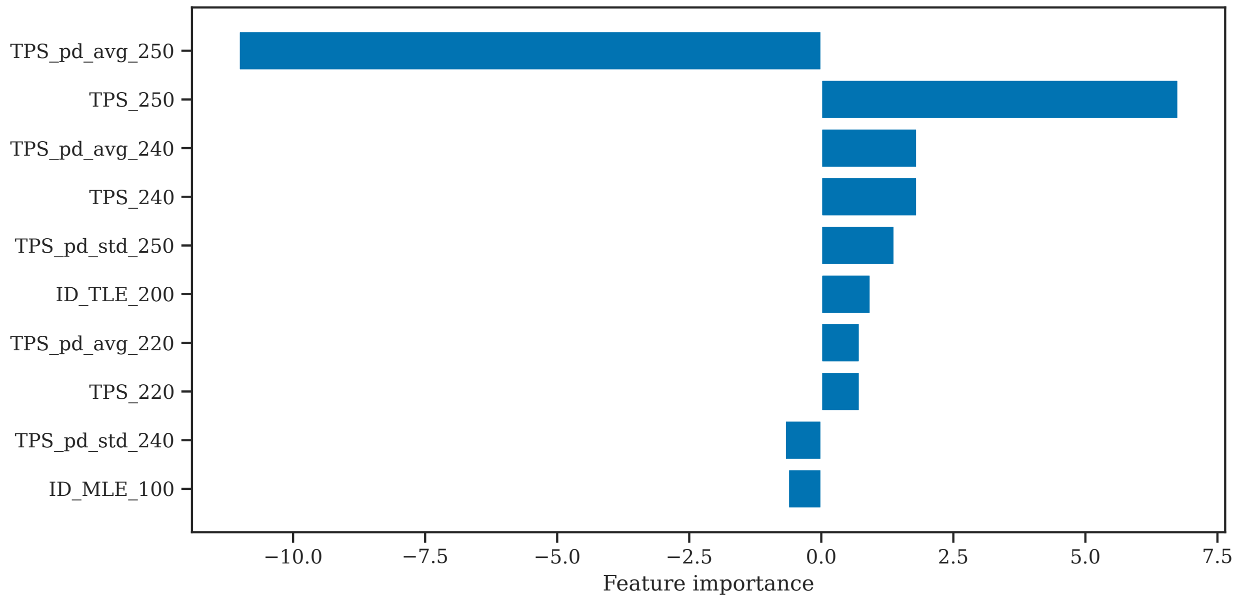

Figure 3.36: Feature importances (i.e. coefficients) for the top 10 most important features of the BWME-enwiki model. We sort the feature importances by their absolute values in descending order.

Furthermore, we show the top 10 feature importances for the TPS, GAD and ID estimator features separately in Figure 3.37, where we see that the TPS features with high values of n are generally more important than the ones with low values of n. From the GAD features in Figure 3.37 (b), we see a different situation to the top 10 features importances for GAD using the WME-enwiki model. In particular, we see that whether or not a point is categorized as singular is important for predicting whether or not a word is polysemous, and the rest of the GAD categories are less relevant. One interesting finding is that the GAD singular features importances were negative; we expected them to be positive, as it would make sense for them to be a positive contribution to whether or not a word is polysemous. Finally, we see that the TLE and TLE ID estimator models yield important features for high neighbourhood values. Similar to the feature importances in Figure 3.33, we note the fact that the feature importances in Figure 3.37 (a) and (c) are more important than the feature importances in Figure 3.37 (b), as noted by the x-axis scales.

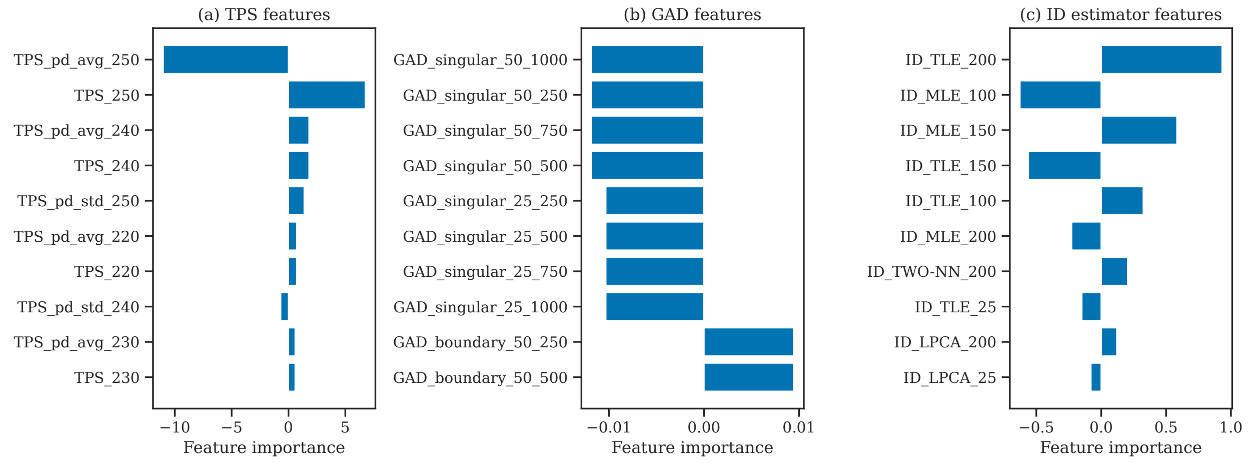

Figure 3.37: Feature importances (i.e. coefficients) of the top 10 TPS, GAD and ID estimator features, using the BWME-enwiki model. We sort the feature importances by their absolute values in descending order.

We have now explained how we trained and evaluated two supervised models for predicting the number of word meanings and whether or not a word is polysemous. Next, we will summarize and conclude the thesis and discuss ideas for future work.

Chapter 4

Summary and Conclusion

In this chapter, we will summarize how we performed the analysis of word embeddings and our findings from Chapter 3 in Section 4.1. Following, we will conclude the thesis in Section 4.2.

4.1 Summary

To summarize the thesis, we first explained how we trained and evaluated our word2vec model in Section 3.1. We showed that our data preprocessing steps and hyperparameter choices led to the training of a relatively high-quality word embedding model that, among other things, understood syntactic and semantic relationships. We also showed the ability of the model to identify underlying concepts of the English language.

Following, we performed a cluster analysis of word embeddings in Section 3.2. We showed that by performing clustering of cluster word embeddings and evaluating the results using internal validation methods, we end up with clusters that intuitively make sense. Additionally, we showed that by clustering word embeddings of distinct word groups, we deepen our understanding of the word embedding manifold. We also used dimensionality reduction methods to visualize results of clustering and showed that the clustering results from the clustering algorithms matched the clusters appearing in the plots.

Next, we investigated the idea of topological polysemy in Section 3.3.1, where we computed the topological polysemy of our word2vec models as well some pre-trained word embedding models. We saw that we were unable to reproduce the results presented in the original paper of topological polysemy, [Jakubowski et al., 2020]. In particular, between the topological polysemy of word embeddings and the true number of word meanings, we got lower correlations and relationships in the opposite direction of the results from [Jakubowski et al., 2020]. Due to the inconsistency in these results, we created a correlation matrix for comparing topological polysemy between our word embedding models and the pre-trained word embeddings. From this, we saw that the fastText model used in the experiments of [Jakubowski et al., 2020] did not correlate well with the rest of the models, signalizing that the word embeddings used in [Jakubowski et al., 2020] can not be compared to other word embeddings. In other words, by applying topological polysemy to other word embeddings than the ones used in [Jakubowski et al., 2020], we can get unexpected and incorrect results. To deepen our understanding of how the topological polysemy was computed, we conducted two experiments by computing topological polysemy of two custom data sets. The two custom data sets consisted of two spheres intersecting in a single data point, in which topological polysemy should, in theory, be able to identify. The first data set was generated without noise, while the second data set had some noise added to it. The two data sets were created using varying dimensions of the spheres, ranging from 2 to 300. We used the sphere data set with no noise in the first experiment and the sphere data set with noise in the second experiment. From the first experiment, we showed that the measure of topological polysemy was unable to identify the intersection point, but rather a spherical area around the intersection point, for all dimensions of the spheres. In the second experiment, we showed that the measure of topological polysemy was unable to identify both the intersection point and the area around it for small sphere dimensions, $d \in \{2, 3\}$, but setting the sphere dimension to 300, we showed that it was possible to identify the intersection point, as it was a significant outlier.

Next, we showed the results of applying the Geometric Anomaly Detection (GAD) algorithm to our word embeddings in Section 3.3.2, where we compared the results to the number of word meanings, as perceived by topological polysemy and WordNet. Using the results from GAD, we were unable to effectively identify polysemous words. We note, however, that this might be due to a misconfiguration of the hyperparameters.

Following, we used ID estimation methods to compute the estimated local ID of word embeddings in Section 3.3.3, where we showed its relation to the number of word meanings perceived by WordNet, which indicated low correlations.

Finally, we created two supervised models estimating the number of word meanings and predicting whether or not a word is polysemous in Section 3.3.4. We used the results from applying topological polysemy, GAD and ID estimation methods to word embeddings to create data used in the supervised models. Furthermore, we used this data to train and evaluate two supervised models for predicting the number of word meanings. We showed a weak correlation between the results from the first supervised model and the number of word meanings and visualized the feature importances in the model. For the second supervised model, we showed the results using confusion matrices, displaying the performance of the model when evaluated to words it has not seen yet, and we showed the feature importances in the model as well. Following, in the next section, we will conclude the thesis.

4.2 Conclusion

To conclude the thesis, we obtain meaningful clusters of word embeddings when we apply them to clustering algorithms and validate the result using internal cluster validation methods. Additionally, topological polysemy does not seem to measure the polysemy of words efficiently and consistently. We note that the topological polysemy method is a relatively new idea, and it requires more work to find out more about the algorithm and its results. In the next chapter, we will discuss our ideas for future work related to this thesis.

Chapter 5

Future Work

In this chapter, we explain our ideas for future work related to this thesis. In particular, we will look at ideas for further developing the analysis of word embeddings, which we employed in Chapter 3.

When analysing word embedding models, we have in this thesis mainly focused on the English language. An interesting approach would be to perform an analysis of word embeddings of various languages, such as the Scandinavian languages, and compare the results to the analysis results using English word embeddings. Additionally, it would be interesting to expand the analysis by focusing on other models than word2vec, such as ELMo [Peters et al., 2018] or BERT [Devlin et al., 2019], particularly for the cluster analysis we performed in Section 3.2.

Furthermore, we attempted to use persistence images (see Section 2.3.3) when creating features using Geometric Anomaly Detection (GAD) in Section 3.3.4, but quickly noticed that we got far too many features and our model had too little capacity to efficiently predict the number of word meanings. To create additional features using GAD, we thought it would be interesting to use a convolutional neural network to learn features from persistence images of persistence diagrams from GAD. Convolutional neural networks are a special type of artificial neural networks and are commonly used to extract features from images [Aggarwal, 2018, Chapter 8].

Next, we thought it would be interesting to use the estimation of local intrinsic dimension (ID) when specifying the manifold dimension k hyperparameter of GAD. By doing so, we could get more realistic results when applying GAD to word embeddings. We note, however, that by setting the manifold dimensionality too high, we get numerical instabilities and overflows when computing the Vietoris–Rips simplicial complexes.

Finally, using the two models for supervised estimation of word meanings, we could improve the training of word embedding models by assigning a unique word embedding for each meaning a word has. Using separate word embeddings for the meanings of a word, we could help to solve the word sense disambiguation (WSD) problem. WSD is the problem of determining which meaning a word has in a particular context (e.g. where the word is in the sentence) [Agirre and Edmonds, 2006, p. 1-2]. We believe that this can be solved if the performance of the supervised models gets to a point where they can effectively separate between monosemous and polysemous words.

Bibliography

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Adams et al., 2016] Adams, H., Chepushtanova, S., Emerson, T., Hanson, E., Kirby, M., Motta, F., Neville, R., Peterson, C., Shipman, P., and Ziegelmeier, L. (2016). Persistence images: A stable vector representation of persistent homology.

[Aggarwal, 2018] Aggarwal, C. C. (2018). *Neural Networks and Deep Learning.* Springer, Cham.

[Agirre and Edmonds, 2006] Agirre, E. and Edmonds, P. (2006). *Word Sense Disambiguation: Algorithms and Applications.* Springer, Dordrecht.

[Allen, 2003] Allen, J. F. (2003). *Natural Language Processing*, page 1218–1222. John Wiley and Sons Ltd., GBR.

[Amsaleg et al., 2019] Amsaleg, L., Chelly, O., Houle, M. E., ichi Kawarabayashi, K., Radovanović, M., and Treeratanajaru, W. (2019). *Intrinsic Dimensionality Estimation within Tight Localities*, pages 181–189. Society for Industrial and Applied Mathematics (SIAM).

[Anderson, 1936] Anderson, E. (1936). The species problem in iris. *Annals of the Missouri Botanical Garden*, 23(3):457–509.

[Andrew Ng and Weiss, 2002] Andrew Ng, M. J. and Weiss, Y. (2002). On spectral clustering: Analysis and an algorithm. In Dietterich, T., Becker, S., and Ghahramani, Z., editors, *Advances in Neural Information Processing Systems*, volume 14, pages 849–856. MIT Press.

[Attardi, 2015] Attardi, G. (2015). Wikiextractor. `https://github.com/attardi/wikiextractor`.

[Aumueller et al., 2021] Aumueller, M., Bernhardsson, E., and Faitfull, A. (2021). Ann-benchmarks. `http://ann-benchmarks.com` (accessed March 30, 2021).

[Bac, 2020] Bac, J. (2020). scikit-dimension - intrinsic dimension estimation in python. `https://github.com/j-bac/scikit-dimension`.

[Bengio et al., 2014] Bengio, Y., Courville, A., and Vincent, P. (2014). Representation learning: A review and new perspectives.

[Bird et al., 2009] Bird, S., Klein, E., and Loper, E. (2009). *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.".

[Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag, Berlin, Heidelberg.

[Boger, 2021] Boger, T. (2021). fastdist: Faster distance calculations in python using numba. `https://github.com/talboger/fastdist`.

[Bojanowski et al., 2017] Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information.

[Caliński and JA, 1974] Caliński, T. and JA, H. (1974). A dendrite method for cluster analysis. *Communications in Statistics - Theory and Methods*, 3:1–27.

[Campello et al., 2013] Campello, R. J. G. B., Moulavi, D., and Sander, J. (2013). Density-based clustering based on hierarchical density estimates. In Pei, J., Tseng, V. S., Cao, L., Motoda, H., and Xu, G., editors, *Advances in Knowledge Discovery and Data Mining*, pages 160–172, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Carter et al., 2010] Carter, K. M., Raich, R., and Hero, A. O. (2010). On local intrinsic dimension estimation and its applications. *Trans. Sig. Proc.*, 58(2):650–663.

[Chazal and Michel, 2021] Chazal, F. and Michel, B. (2021). An introduction to topological data analysis: fundamental and practical aspects for data scientists.

[Datafiniti, 2017] Datafiniti (2017). Food ingredient lists. `https://data.world/datafiniti/food-ingredient-lists` (accessed January 15, 2021).

[Davies and Bouldin, 1979] Davies, D. L. and Bouldin, D. W. (1979). A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227.

[Devlin et al., 2019] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159.

[Dupras, 2014] Dupras, V. (2014). num2words library - convert numbers to words in multiple languages. `https://github.com/savoirfairelinux/num2words`.

[Edelsbrunner and Harer, 2010] Edelsbrunner, H. and Harer, J. (2010). *Computational Topology: An Introduction*. American Mathematical Society.

[Ester et al., 1996] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press.

[Facco et al., 2017] Facco, E., d'Errico, M., Rodriguez, A., and Laio, A. (2017). Estimating the intrinsic dimension of datasets by a minimal neighborhood information. *Scientific Reports*, 7(1):12140.

[Fellbaum, 1998] Fellbaum, C. (1998). Wordnet: An electronic lexical database. `https://wordnet.princeton.edu/`.

[Feng et al., 2020] Feng, Y., Major, S., and Sievert, S. (2020). rainwoodman/sharedmem 0.3.8.

[Fisher, 1936] Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188.

[Fox, 2015] Fox, J. (2015). *Applied Regression Analysis and Generalized Linear Models*. SAGE Publications.

[Fukunaga and Olsen, 1971] Fukunaga, K. and Olsen, D. (1971). An algorithm for finding intrinsic dimensionality of data. *IEEE Transactions on Computers*, C-20(2):176–183.

[GeoNames, 2005] GeoNames (2005). Geonames. `https://www.geonames.org` (accessed May 2, 2021).

[Gladkova et al., 2016] Gladkova, A., Drozd, A., and Matsuoka, S. (2016). Analogy-based detection of morphological and semantic relations with word embeddings: what works and what doesn't. In *Proceedings of the NAACL Student Research Workshop*, pages 8–15, San Diego, California. Association for Computational Linguistics.

[Google Code Archive, 2013] Google Code Archive (2013). Google code archive - word2vec. `https://code.google.com/archive/p/word2vec` (accessed April 2, 2021).

[Grave et al., 2018] Grave, E., Bojanowski, P., Gupta, P., Joulin, A., and Mikolov, T. (2018). Learning word vectors for 157 languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*.

[Guo et al., 2020] Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., and Kumar, S. (2020). Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*.

[Haro et al., 2008] Haro, G., Randall, G., and Sapiro, G. (2008). Translated poisson mixture model for stratification learning. *International Journal of Computer Vision*, 80(3):358–374.

[Harris et al., 2020] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., Fernández del Río, J., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585:357–362.

[Hunter, 2007] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95.

[Jakubowski et al., 2020] Jakubowski, A., Gašić, M., and Zibrowius, M. (2020). Topology of word embeddings: Singularities reflect polysemy.

[James et al., 2013] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications in R*. Springer.

[Joblib Development Team, 2021] Joblib Development Team (2021). Joblib: running python functions as pipeline jobs.

[Joe H. Ward, 1963] Joe H. Ward, J. (1963). Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244.

[Jolliffe, 2002] Jolliffe, I. (2002). *Principal component analysis*. Springer Verlag, New York.

[Kaufman and Rousseeuw, 1990] Kaufman, L. and Rousseeuw, P. (1990). *Partitioning Around Medoids (Program PAM)*, chapter 2, pages 68–125. John Wiley & Sons, Ltd.

[Lee et al., 2015] Lee, S., Campadelli, P., Casiraghi, E., Ceruti, C., and Rozza, A. (2015). Intrinsic dimension estimation: Relevant techniques and a benchmark framework. *Mathematical Problems in Engineering*, 2015:759567.

[Levina and Bickel, 2004] Levina, E. and Bickel, P. J. (2004). Maximum likelihood estimation of intrinsic dimension. In *Proceedings of the 17th International Conference on Neural Information Processing Systems*, NIPS'04, page 777–784, Cambridge, MA, USA. MIT Press.

[Levy et al., 2015] Levy, O., Goldberg, Y., and Dagan, I. (2015). Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225.

[Manandhar and Klapaftis, 2009] Manandhar, S. and Klapaftis, I. (2009). SemEval-2010 task 14: Evaluation setting for word sense induction & disambiguation systems. In *Proceedings of the Workshop on Semantic Evaluations: Recent Achievements and Future Directions (SEW-2009)*, pages 117–122, Boulder, Colorado. Association for Computational Linguistics.

[McInnes, 2016] McInnes, L. (2016). How hdbscan works. `https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html` (accessed: 12th of April 2021).

[McInnes, 2018] McInnes, L. (2018). How umap works. `https://umap-learn.readthedocs.io/en/latest/how_umap_works.html` (accessed: 1st of March 2021).

[McInnes et al., 2017] McInnes, L., Healy, J., and Astels, S. (2017). hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11):205.

[McInnes et al., 2018] McInnes, L., Healy, J., and Melville, J. (2018). UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *ArXiv e-prints*.

[McInnes et al., 2018] McInnes, L., Healy, J., Saul, N., and Grossberger, L. (2018). Umap: Uniform manifold approximation and projection. *The Journal of Open Source Software*, 3(29):861.

[Mikolov, 2013a] Mikolov, T. (2013a). tmikolov/word2vec - demo-phrases.sh. `https://github.com/tmikolov/word2vec/blob/master/demo-phrases.sh` (accessed September 14, 2020).

[Mikolov, 2013b] Mikolov, T. (2013b). tmikolov/word2vec - word2phrase.c. `https://github.com/tmikolov/word2vec/blob/master/word2phrase.c` (accessed September 14, 2020).

[Mikolov, 2013c] Mikolov, T. (2013c). tmikolov/word2vec - word2vec.c. `https://github.com/tmikolov/word2vec/blob/master/word2vec.c` (accessed September 14, 2020).

[Mikolov et al., 2013a] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space.

[Mikolov et al., 2013b] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality.

[Mikolov et al., 2013c] Mikolov, T., Yih, W.-t., and Zweig, G. (2013c). Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, Atlanta, Georgia. Association for Computational Linguistics.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition.

[Oudot, 2015] Oudot, S. (2015). *Persistence Theory: From Quiver Representations to Data Analysis*. American Mathematical Society.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Édouard Duchesnay (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830.

[Pennington, 2014] Pennington, J. (2014). Glove: Global vectors for word representation. `https://nlp.stanford.edu/projects/glove` (accessed April 2, 2021).

[Pennington et al., 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

[Peters et al., 2018] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations.

[Rokach and Maimon, 2005] Rokach, L. and Maimon, O. (2005). *Clustering Methods*, pages 321–352. Springer US, Boston, MA.

[Rong, 2016] Rong, X. (2016). word2vec parameter learning explained.

[Rouvreau, 2021] Rouvreau, V. (2021). Cython interface. In *GUDHI User and Reference Manual*. GUDHI Editorial Board, 3.4.1 edition.

[Scikit-learn contrib, 2021] Scikit-learn contrib (2021). scikit-learn-extra - a set of useful tools compatible with scikit-learn. `https://github.com/scikit-learn-contrib/scikit-learn-extra`.

[Sculley, 2010] Sculley, D. (2010). Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 1177–1178, New York, NY, USA. Association for Computing Machinery.

[Simon Zhang and Wang, 2020] Simon Zhang, M. X. and Wang, H. (2020). Gpu-accelerated computation of vietoris-rips persistence barcodes.

[Social Security Administration, 2019] Social Security Administration (2019). Popular baby names. https://www.ssa.gov/oact/babynames/limits.html (accessed January 14, 2021).

[Stolz et al., 2020] Stolz, B. J., Tanner, J., Harrington, H. A., and Nanda, V. (2020). Geometric anomaly detection in data. *Proceedings of the National Academy of Sciences*, 117(33):19664–19669.

[TensorFlow team, 2020] TensorFlow team (2020). Making new layers and models via subclassing. https://www.tensorflow.org/guide/keras/custom_layers_and_models (accessed September 14, 2020).

[Tralie et al., 2018] Tralie, C., Saul, N., and Bar-On, R. (2018). Ripser.py: A lean persistent homology library for python. *The Journal of Open Source Software*, 3(29):925.

[Triki, 2021] Triki, J. F. (2021). Analysis of word embeddings: A clustering and topological approach. https://github.com/JonasTriki/masters-thesis-ml.

[U.S. Census Bureau, 2010] U.S. Census Bureau (2010). Frequently occurring surnames from the 2010 census. https://www.census.gov/topics/population/genealogy/data/2010_surnames.html (accessed January 14, 2021).

[van Kooten, 2016] van Kooten, P. (2016). contractions. https://github.com/kootenpv/contractions.

[Virtanen et al., 2020] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

[Walkowiak and Gniewkowski, 2019] Walkowiak, T. and Gniewkowski, M. (2019). Evaluation of vector embedding models in clustering of text documents. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2019)*, pages 1304–1311, Varna, Bulgaria. INCOMA Ltd.

[Waskom, 2021] Waskom, M. L. (2021). seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021.

[Wikimedia, 2021] Wikimedia (2021). Wikimedia downloads. `https://dumps.wikimedia.org` (accessed February 2, 2021).

[Zibrowius, 2021] Zibrowius, M. (2021). Private Communication.