Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last years, the amount of available data has been increasing permanently. Companies in most industries started to realise that their data contains a lot of useful information and that they can use it to optimise their processes. Also research benefits a lot from the increasing availability of data. Machine learning algorithms use data to learn various tasks, e.g. how to recognize a person by their face. These algorithms not only do well learning such tasks, but they even start performing better than humans. An example that shows the increasing amount of available data is the amount of websites as shown in figure 1.1. It took the world-wide web 23 years (1989-2012) to reach one billion websites where the next billion websites only needed six years (2012-2018). Another domain where similar changes can be observed is image data, that is accessible though different platforms such as Facebook and Instagram. By having a large amount of data, machine learning algorithms can perform very well.

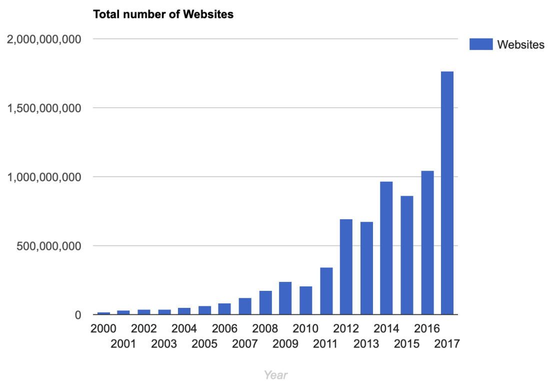

FIGURE 1.1: The increasing amount of website is only one example that describes that enourmous increasing availability of data that also appears for other data.

However, one problem of many state-of-the-art machine learning algorithms is that they can solve one task well, but only this task. Imagine a classifier that can distinguish between different animals. It may have learned to perform very well and can differentiate different animals such as a leopard and a tiger. The classifier learned some certain representations that identify the animals occurately. Nevertheless, the classifier only learned to describe animals. An image such as the one shown in 1.2 may be able to trick the classifier by being evaluated as a leopard.

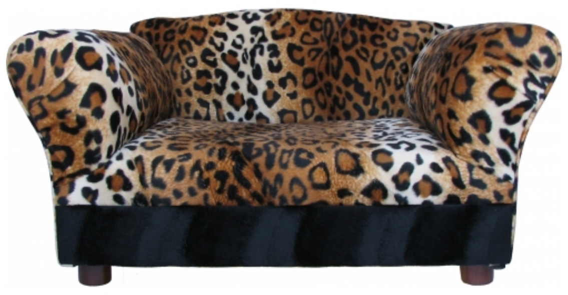

Figure 1.2: When a machine learning algorithm fails to transfer knowledge to other tasks and / or domains, a classifier may recognize a couch as a leopard.

Where we can give general information about existing algorithms, their runtime and their accuracy, this thesis aims to design algorithms that are adapted to certain text and image data and thus are able to perform more efficiently on the data than existing general solutions. Also, these algorithms are designed to be adaptable to other clustering tasks within the same data domain.

The here proposed algorithms perform clustering tasks, i.e. they divide existing data into different subspaces. Imagine different animals, one potential subspace could be pets where on the other side there are wild animals. The proposed clustering algoriths belong to a specific family that will be introduced in chapter 2.

Chapter 2

Background Theory

2.1 Data-driven Algorithm Design

This increasing amount of data allows us improve the learning capabilities of machines. We know how well existing algorithms perform for any kind of data and which runtime guarantees they have. However, the algorithms' guarantees are general observations and can vary a lot between different data. In many real-world applications the data does not vary that much, e.g. the data for clustering websites into different types may vary quite much on a yearly base, but as this task gets exectued thousands of times each second for certain search algorithms, the data will not change much. By assuming a static context, it is then possible to leverage the context to improve the algorithmic results, e.g. say you want to cluster person data for different genders. By having this a-priori information, you can use a k-means clustering algorithm with $k = 3$ in order to differentiate between female, male and non-binary people.

However, such observations are mostly not that trivial and often require more effort in order to obtain useful a-priori information. In order to cluster financial standing, one could imagine seeing different clusters depending on the age or the education. But how many clusters would result here? The data has to be processed and evaluated for different values in this case.

2.2 Transfer Learning

Once our algorithm performs well for our data and our tasks, we then want to transfer the gained knowledge to different tasks. Say the algorithm already learned how to differentiate images of the handwritten digits zero, one and two, the same algorithm should then be able to apply the gained knowledge to distinguish between other handwritten digits too. The gained knowledge is some kind of learned data, that can for example be the feature representation of a Convolutional Neural Network, where a potential goal can be to transfer the representation knowledge to another classification task.

For clustering tasks learned knowledge could be a number of clusters, a good feature representation for the input data or other useful information that allows performing similar clustering tasks better by transferring the knowledge.

2.3 Linkage-based hierarchical clustering

This thesis focuses on agglomerative hierarchical clustering, i.e. clustering algorithms that merge clusters starting from each cluster as its own point until all points belong to the same cluster. At each iteration the clusters with the closest distance are merged together. As there are various clustering algorithms, there also are various distance measurements. One way of describe the distance between two clusters, say X and Y is by defining a linkage between them. There are three different methods to do so.

2.3.1 Single Linkage

Single linkage defines a distance between two clusters X and Y as the distance between the two nearest points of these clusters (see equation 2.1).

$$d_{SL}(X,Y) = \min_{x \in X, y \in Y} d(x,y) \tag{2.1}$$

2.3.2 Complete Linkage

Complete linkage defines a distance between two clusters X and Y as the distance between the two farthest points of these clusters (see equation 2.2).

$$d_{CL}(X,Y) = \max_{x \in X, y \in Y} d(x,y) \tag{2.2}$$

2.3.3 Average Linkage

Average linkage defines a distance between two clusters X and Y as the average distance between all points $x \in X$ and all points $y \in Y$ (see equation 2.3).

$$d_{AL}(X,Y) = \frac{1}{\|X\|\|Y\|} \sum_{x \in X, y \in Y} d(x,y) \tag{2.3}$$

2.3.4 Effects of different linkage strategies

Depending on the linkage strategy, the pairwise distances between all N clusters $C_1, ..., C_N$ will be different. As the clustering algorithm merges the closest pair of clusters in each iteration, the merging clusters C_i and C_j with $i, j \in 1, ..., N$ might vary as shown in figure 2.1, where ten clusters $C_0, ..., C_9$ get clustered with bottom-up hierarchical clustering using the Euclidean distance as distance $d(x, y)$ to calculate the pairwise distance according to the three mentioned linkage strategies.

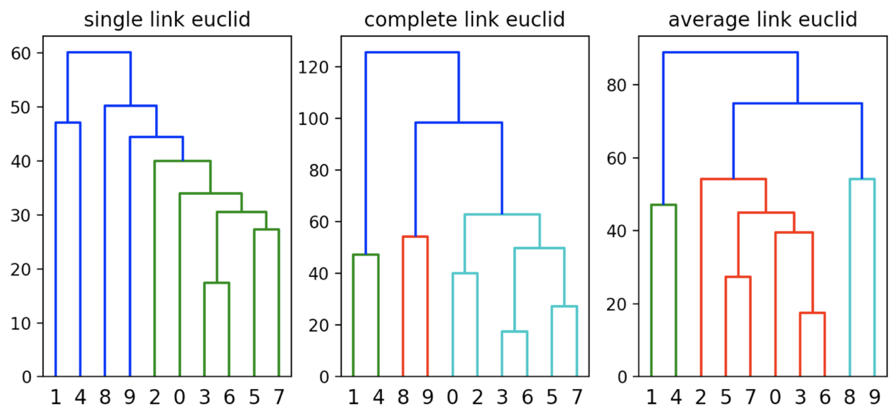

FIGURE 2.1: Different distance measurements often result in different merges for bottom-up hierarchical clustering algorithms. The three discussed linkage strategies result in three different clusterings.

As different points are merged together, this also means that the clustering may have a different quality. This thesis compares the clusterings' quality for different data by introducing algorithms to efficiently determine the quality not only for these linkage strategies but also for their linear combinations.

Chapter 3

Related Work

Balcan et al. proposed the two infinite families to interpolate between different linkage strategies [1], such as shown in equations 3.1 and 3.2.

$$\mathcal{A}_1 = \left\{ \left(\min_{u \in A, v \in B} (d(u,v))^\alpha + \max_{u \in A, v \in B} (d(u,v))^\alpha \right)^{1/\alpha} \middle| \alpha \in \mathbb{R} \cup \{\infty, -\infty\} \right\} \tag{3.1}$$

Equation 3.1 shows a distances in the range between single linkage ($\alpha = -\infty$) and complete linkage ($\alpha = \infty$). Balcan et al. also show that $\mathbb{R} \cup \{\infty, -\infty\}$ contains a maximum of $O(n^8)$ different intervals, where each interval $[\alpha_{lo}, \alpha_{hi}]$ represents a different merging behavior.

$$\mathcal{A}_2 = \left\{ \left(\frac{1}{\|A\| \|B\|} \sum_{u \in A, v \in B} (d(u,v))^\alpha \right)^{1/\alpha} \middle| \alpha \in \mathbb{R} \cup \{\infty, -\infty\} \right\} \tag{3.2}$$

Equation 3.2 will also result in single linkage for $\alpha = -\infty$ and complete linkage for $\alpha = \infty$. In addition to that, the family $\mathcal{A}_2$ also contains the definition of average linkage ($\alpha = 0$). However, the guarantee for maximum $O(n^8)$ intervals does not apply to this family. A formal guarantee will be $O(n^{n-1})$, but this thesis will show that the experimental results are much better than the actual formal guarantee.

In addition to the families $\mathcal{A}_1$ and $\mathcal{A}_2$, this thesis will also propose another family to efficiently interpolate between single, average and complete linkage in section **??**.

Chapter 4

α-Linkage

We define α as the parameter with which the output of an algorithm is weighted. In this chapter we propose different distance measures depending on the weight parameter α that allows us interpolating between different linkage strategies in a way similar to the proposed by Balcan et al. [1], where a infinite interval was proposed.

To have a real application, we need to have a finiteset of intervals. So we interpolate between one algorithm with $\alpha = 0$ and another algorithm with $\alpha = 1$, where for $\alpha = 0$ the result will be the result of algorithm 1 and for $\alpha = 1$ the result of algorithm 2.

4.1 Linear Interpolation between two different linkage strategies

Interpolating between two of the three mentioned linkage strategies results in three different algorithmic settings. In the first setting we are using the single linkage distance $d_{SL}(X, Y)$ and the complete linkage distance $d_{CL}(X, Y)$. By combining the two distances we can create a linear model that ranges from $\alpha = 0$ (single linkage) to $\alpha = 1$ (complete linkage) resulting in equation 4.1.

$$\begin{aligned} d_{SC}(X, Y, \alpha) &= (1 - \alpha) \cdot d_{SL}(X, Y) + \alpha \cdot d_{CL}(X, Y) \\ &= (1 - \alpha) \min_{x \in X, y \in Y} d(x, y) + \alpha \max_{x \in X, y \in Y} d(x, y) \end{aligned} \tag{4.1}$$

Equivalently we can interpolate between the single linkage distance $d_{SL}(X, Y)$ and the average linkage distance $d_{AL}(X, Y)$ instead of the complete linkage distance $d_{CL}(X, Y)$ for $\alpha = 1$ resulting in equation 4.2.

$$\begin{aligned} d_{SA}(X,Y,\alpha) &= (1-\alpha)\cdot d_{SL}(X,Y) + \alpha \cdot d_{AL}(X,Y) \\ &= (1-\alpha) \min_{x \in X, y \in Y} d(x,y) + \alpha \frac{1}{\|X\|\|Y\|} \sum_{x \in X, y \in Y} d(x,y) \end{aligned} \tag{4.2}$$

The last of the three settings describes the interpolation between the average linkage distance $d_{AL}(X,Y)$ and the complete linkage distance $d_{CL}(X,Y)$ resulting in equation 4.3.

$$\begin{aligned} d_{AC}(X,Y,\alpha) &= (1-\alpha)\cdot d_{AL}(X,Y) + \alpha \cdot d_{CL}(X,Y) \\ &= (1-\alpha) \frac{1}{\|X\|\|Y\|} \sum_{x \in X, y \in Y} d(x,y) + \alpha \max_{x \in X, y \in Y} d(x,y) \end{aligned} \tag{4.3}$$

4.2 Proposed Algorithms

Our goal is to find an algorithm that determines all different behavior depending on different values of α. To do so, we propose different algorithms. The goal of the first algorithm is to divide the interval of $\alpha \in [\alpha_{lo}, \alpha_{hi}]$ to subintervals where the behavior is consistent within each interval.

```
**Data:** input data $p_1, ..., p_N$, initial states $st$
**Result:** $k$ intervals $[\alpha_0, \alpha_1], ..., [\alpha_{k-1}, \alpha_k]$
for $iteration \leftarrow 1$ to $N-1$ do
    foreach $state\ s \in st$ do
        remove state $s$;
        $cand_1, cand_2 \leftarrow$find merge candidates for $s.\alpha_{lo}$ and $s.\alpha_{hi}$;
        if $cand_1 == cand_2$ then
            $ms \leftarrow$ merge $cand_1$;
            add state $ms$ with interval $[\alpha_{lo}, \alpha_{hi}]$ to the end of $st$;
        else
            $\alpha_{split} \leftarrow$ calculate split;
            $s_1 \leftarrow$ merge $cand_1$;
            $s_2 \leftarrow$ merge $cand_2$;
            add state $s_1$ with interval $[\alpha_{lo}, \alpha_{split}]$ to the end of $st$;
            add state $s_2$ with interval $[\alpha_{split}, \alpha_{hi}]$ to the end of $st$;
        end
    end
end
```

Algorithm 1: We calculate all from splits resulting different intervals between α_{lo} and α_{hi}, merge the resulting clusters and do so until each state contains only one cluster with all points.

Starting from an interval $\alpha \in [\alpha_{lo}, \alpha_{hi}]$, we calculate the merging clusters by $\min_{X,Y} d(X,Y,\alpha)$ for both the minimum α_{li} and the maximum α_{hi} of the interval. In case both values of α

return the same pair of merging clusters X and Y, we merge X and Y. In case the values of α_{lo} and α_{hi} lead to different merges, we can calculate a value α_{split} where we know that for values of $\alpha \in [\alpha_{lo}, alpha_{split})$ we merge the clusters found for $\min\limits_{X,Y} d(X, Y, \alpha_{lo})$ and for values of $\alpha \in [\alpha_{split}, alpha_{hi}]$ we merge the clusters found for $\min\limits_{X,Y} d(X, Y, \alpha_{hi})$. In order to calculate the value of α_{split} we can equalize the distance functions of the merges of clusters X, Y and the clusters A, B (say α_{lo} leads to merging X and Y and α_{hi} leads to merging A and B) as seen in equation 4.4.

$$d(X, Y, \alpha_{split}) = d(A, B, \alpha_{split}) \tag{4.4}$$

Applying 4.4 to a concrete example of distance functions leads to a concrete calculation for the value of α_{split}. Equation 4.5 shows the calculation for the in equation 4.1 introduced d_{SC}.

$$\begin{gathered}
d_{SC}(X, Y, \alpha_{split}) = d_{SC}(A, B, \alpha_{split}) \\
(1-\alpha_{split}) \min_{x\in X, y\in Y} d(x,y) + \alpha_{split} \max_{x\in X, y\in Y} d(x,y) = \\
= (1-\alpha_{split}) \min_{a\in A, b\in B} d(a,b) + \alpha_{split} \max_{a\in A, b\in B} d(a,b) \\
(-\alpha_{split}) \min_{x\in X, y\in Y} d(x,y) + \alpha_{split} \max_{x\in X, y\in Y} d(x,y) + \alpha_{split} \min_{a\in A, b\in B} d(a,b) - \alpha_{split} \max_{a\in A, b\in B} d(a,b) = \\
= - \min_{x\in X, y\in Y} d(x,y) + \min_{a\in A, b\in B} d(a,b) \\
\alpha_{split}(- \min_{x\in X, y\in Y} d(x,y) + \max_{x\in X, y\in Y} d(x,y) + \min_{a\in A, b\in B} d(a,b) - \max_{a\in A, b\in B} d(a,b)) = \\
= - \min_{x\in X, y\in Y} d(x,y) + \min_{a\in A, b\in B} d(a,b) \\
\alpha_{split} = \frac{- \min_{x\in X, y\in Y} d(x,y) + \min_{a\in A, b\in B} d(a,b)}{- \min_{x\in X, y\in Y} d(x,y) + \max_{x\in X, y\in Y} d(x,y) + \min_{a\in A, b\in B} d(a,b) - \max_{a\in A, b\in B} d(a,b)}
\end{gathered} \tag{4.5}$$

After knowing the exact consistent range, we split the clusters for the different states and then calculate the merge candidates for the start and the end of the new intervals again. We can show the different possible merges in a tree of executions, where each node represents one interval $[\alpha_{lo}, \alpha_{hi}]$ where the same clusters get merged (see figure 4.1).

We perform the described procedure for iterations $i = count(points) - 1$ times until only one cluster containing all points is left. All the leaf nodes in the resulting tree of executions then represent one interval $[\alpha_{lo}, \alpha_{hi}]$ where the clustering is consistent within the interval and each interval contains a different clustering.

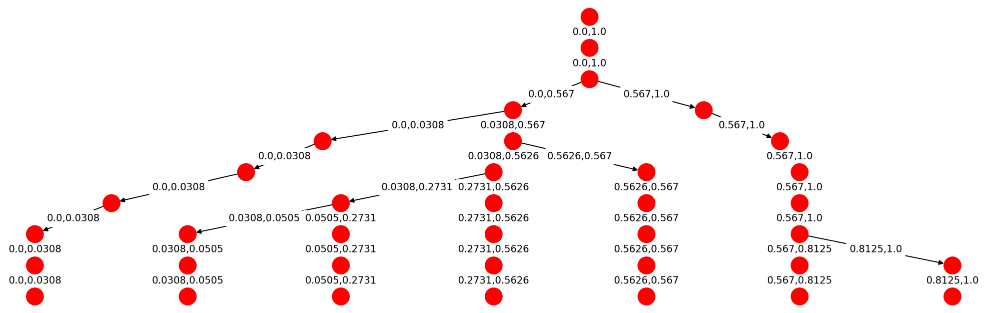

FIGURE 4.1: Different values of α lead to different merges. We calculate a tree where we start with the entire range $[\alpha_{lo}, \alpha_{hi}]$ and split the interval into all different subintervals with consistent merges.

However just calculating the split values in a range $[\alpha_{lo}, \alpha_{hi}]$ does not necessarily yield to the best possible solution. One of these examples is demonstrated in figure 4.2, where the blue line for the constant value of d will not be considered, only the lines $\alpha \in [\alpha_{lo}, alpha_{split})$ (red) and $\alpha \in [\alpha_{split}, alpha_{high}]$ (black) will be.

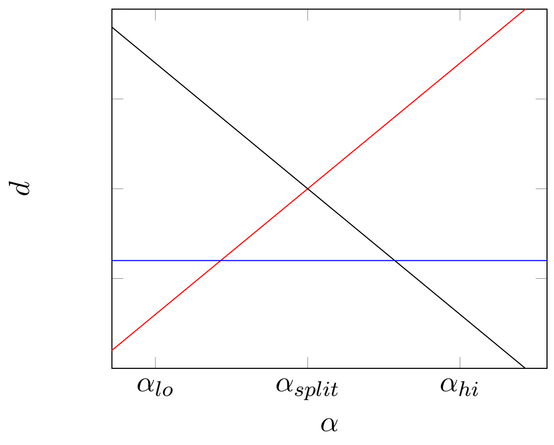

FIGURE 4.2: Simply calculating the split values between the start and the end value of the range $[\alpha_{lo}, \alpha_{hi}]$ will not necessarily lead to the optimal values. By doing so, the blue line (constant d value) will not be considered.

In order to solve this, we can recursively check each resulting interval again if it contains different merging behaviors.

By calculating the split points recursively, the example in figure 4.3 will result in the intervals $[\alpha_{lo}, \alpha_{s_1}]$, $[\alpha_{s_1}, \alpha_{s_2}]$, $[\alpha_{s_2}, \alpha_{s_3}]$ and $[\alpha_{s_3}, \alpha_{s_{hi}}]$. The optimal distance between α_{s_1} and α_{s_3} is covered now, but the results contain one unncessary interval as α_{s_2} still splits two intervals. The algorithm can check if older splits are still relevant, however the runtime cost to do so will be more expensive than carrying one additional interval

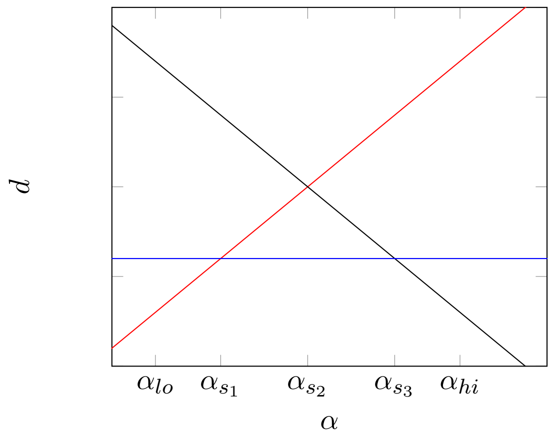

Figure 4.3: Simply calculating the split values between the start and the end value of the range $[\alpha_{lo}, \alpha_{hi}]$ will not necessarily lead to the optimal values. By doing so, the blue line (constant d value) will not be considered.

with the same distance. We can use this knowledge and adapt algorithm 1.

```
Data: input data $p_1, ..., p_N$, initial states $st$
Result: $k$ intervals $[\alpha_0, \alpha_1], ..., [\alpha_{k-1}, \alpha_k]$
for $iteration \leftarrow 1$ to $N - 1$ do
    foreach $state\ s \in st$ do
        remove state $s$;
        ranges $\leftarrow$ find ranges between $s.\alpha_{lo}$ and $s.\alpha_{hi}$;
        foreach $range\ r \in ranges$ do
            $cand \leftarrow$ candidate for range;
            $ms \leftarrow$ merge $cand$;
            add state $ms$ with range $r$ to the end of $st$;
        end
    end
end
```

Algorithm 2: By calculating the split points between α_{lo} and α_{hi} recursively, we ensure that no optimal interval is left out.

As experimental results turn out to need a lot of memory (up to $\approx$ 20 GB for 300 points and 20,000 states), we want to adapt algorithm 2 so that it uses less memory. The memory usage scales relative to the amount of currently in-memory stored states, so the goal is to reduce these. As the amount of states is much larger than the amound of iterations, we calculate and evaluate the leave nodes of the tree and keep the alternative merges stored. This results in algorithm 3.

```
Data: input data $p_1, ..., p_N$, initial states $st$
Result: $k$ intervals $[\alpha_0, \alpha_1], ..., [\alpha_{k-1}, \alpha_k]$
while $\|st\| > 0$ do
    foreach state $s \in st$ do
        remove state $s$;
        if $s$ is final then
            evaluate $s$;
        else
            ranges $\leftarrow$ find ranges between $s.\alpha_{lo}$ and $s.\alpha_{hi}$;
            foreach range $r \in ranges$ do
                $cand \leftarrow$ candidate for range;
                $ms \leftarrow$ merge $cand$;
                add state $ms$ with range $r$ to the beginning of $st$;
            end
        end
    end
end
```

Algorithm 3: Instead of calculating the nodes layerwise, this algorithm works pathwise, i.e. it goes down one path of a tree to a leaf node and evaluates it before continuing with the next split. This approach needs much less memory than the previous algorithms and has about the same runtime as shown in figure 4.4.

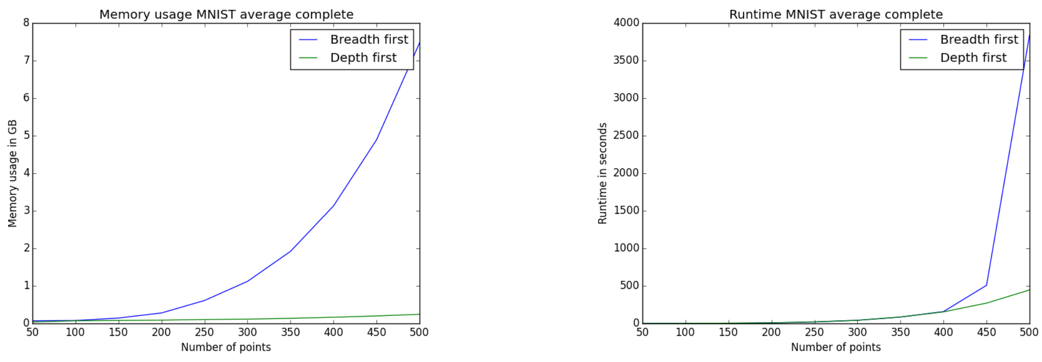

FIGURE 4.4: The depth first implementation needs less memory and also has a better runtime compared to the breadth first implementation.

Instead of merging iteratively and steadily shrinking the intervals, we propose an algorithm with a geometric motivation. We are again evaluating an interval $[\alpha_{lo}, \alpha_{hi}]$, but we interprete the different merges as linear functions depending on α. We can start by calculating the merge candidate for the start value α_{lo} and calculate the next intersection that will yield to the next merge. By calculating all the intersections of linear functions, we can also determine all the different intervals for the range $[\alpha_{lo}, \alpha_{hi}]$, where different merging behaviors occure. Algorithm 4 describes this procedure.

```
Data: input data $p_1, ..., p_N$, start value $\alpha_{lo}$, end value $\alpha_{hi}$
Result: $k$ intervals $[\alpha_0, \alpha_1], ..., [\alpha_{k-1}, \alpha_k]$
$\alpha \leftarrow \alpha_{lo}$;
linear function $lf \leftarrow$ get lf for alpha;
while $\alpha < \alpha_{hi}$ do
    $\alpha_{new} \leftarrow$ calculate next split for $\alpha$;
    $lf \leftarrow$ get lf for $\alpha_{new}$;
    $\alpha \leftarrow \alpha_{new}$
end
```

Algorithm 4:

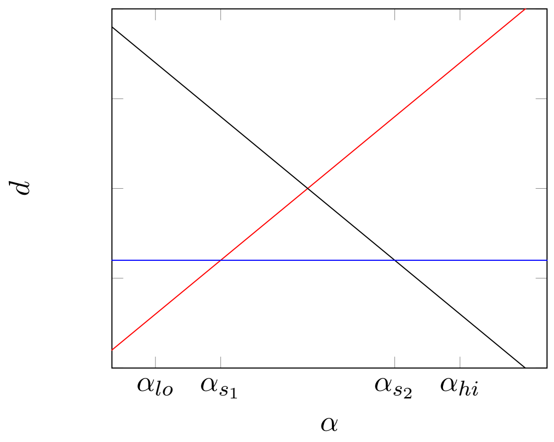

FIGURE 4.5: Simply calculating the split values between the start and the end value of the range $[\alpha_{lo}, \alpha_{hi}]$ will not necessarily lead to the optimal values. By doing so, the blue line (constant d value) will not be considered.

4.3 Performance Optimizations

In order to have real-world applications, the proposed algorithms should run in an efficient way, i.e. it should not take the α-linkage algorithms too much time to run. A first python implementation took days to run, but switching to C++ and using its advantages took down the runtime to hours. However, there are more optimization methods that we used in order to improve the runtime.

4.3.1 Dynamic Programming

One of the most time-consuming parts was the calculating of the distances. For each pair of clusters C_i, C, j the distance had to be calculated for each clustering state. We optimized this by using dynamic programming and stored the distance matrices D_{lower} and D_{upper} for each state. The naming results from the different interpolation settings where we interpolate from one linkage distance (lower) to another linkage distance (upper), e.g. the setting in equation 4.1 describes the interpolation from single linkage (lower) to complete linkage (upper). In this example we then store the pairwise distances for both single linkage and complete linkage and in order to find the merge candidates we have to do iterate over the distance matrices instead of calculating the distances over and over again. When we merge two clusters, we then update the distance matrices for the given state. Table 4.1 shows an example for the pairwise distances of clusters i and j.

j\i	0	1	2	3	4
0	0	1.243	1.512	2.468	5.1243
1	1.243	0	2.443	3.1412	4.443
2	1.512	2.443	0	3.8988	6.827
3	2.468	3.1412	3.8988	0	5.72
4	5.1243	4.443	6.827	5.72	0

Table 4.1: Storing the pairwise distances of all clusters avoids calculating the distances over and over again.

One observation that we can make is that the matrix has a lot of redundant values, because $D(i,j) = D(j,i)$. Removing these rendundant values will result in a trade-off between copying and indexing costs and will be discussed in the following section. Another optimization we can do is storing the indices of the active clusters, i.e. the clusters that can get merged. Once two clusters got merged, they cannot be merged any further, only the resulting cluster can. So we then do not have to consider the old clusters anymore and can remove them from the set of active indicies. This allows us to find the merge candidates faster as the pool of candidates gets smaller.

4.3.2 Trade-Off between Copying and Indexing Costs

Currently we can access the costs for a pair of clusters C_i and C_j through $D[i,j]$ or $D[i + j * width]$ for flattened matrices. These indices are very easy to calculate. In order to remove the redundant values from the distance matrix we remove all values below the diagonal as shown in table 4.2.

In addition to that we can also remove the diagonal values as they represent the distances between the same clusters and are thus always zero. This results in table 4.3.

The matrices are now smaller, so they need less memory. In the example, we changed a matrix of the size 25 to a matrix of the size 10. In general a matrix of the size nxn will

j\i	0	1	2	3	4
0	0	1.243	1.512	2.468	5.1243
1		0	2.443	3.1412	4.443
2			0	3.8988	6.827
3				0	5.72
4					0

TABLE 4.2: Storing the pairwise distances of all clusters avoids calculating the distances over and over again.

j\i	0	1	2	3	4
0		1.243	1.512	2.468	5.1243
1			2.443	3.1412	4.443
2				3.8988	6.827
3					5.72
4					

TABLE 4.3: Storing the pairwise distances of all clusters avoids calculating the distances over and over again.

be compressed to a matrix of the size $\frac{n^2-n}{2}$. The lower amount of needed memory also results in less copying costs that will lead to a better runtime. However, the indexing is not as easy anymore. For easier storage, we again work with flattened matrices, the indexing for the resulting list is shown in equation 4.6.

$$index(i,j) = \frac{width * (width - 1)}{2} - \frac{(width - j) * (width - j - 1)}{2} + i - j - 1 \quad (4.6)$$

Calculating this index in a nested loop is very expensive, however we calculate the part that does not depend on i in the outer loop and thus only need to add i in the inner loop. This does not only yield to a lower memory usage of $\approx 30\%$, but also increases the runtime by TODO.

4.3.3 Implementation-specific Optimizations

In order to optimize the implementation even further, we will have a look into the implementation. One optimization that already was briefly described is the flatterning of the matrices, so the resulting list will be one-dimensional and can be iterated easier and faster.

Another observation is that copy operations are computationally expensive, so we avoid them as much as possible. In the described algorithms (1, 2 and 3) we removed a state from the list of states and added other states. In an optimized way, we do not remove the state and just overwrite the state with the resulting state. Once there are splits in the current interval, the state gets overwritten and additional states get added to the list.

We can also optimize the way of updating the distance matrices. Instead of adding new clusters there for a merge of clusters i and j we update the distances of i to all active clusters with the distances of the resulting cluster. The distances of the cluster j will not be considered for merges anymore as the index j gets removed from the active indices. This has the advantage that the size of the distance matrices will not increase after merges.

Also, the data types make an important contribution to the memory usage. Instead of using double precision floating point values, single precision is enough to clearly identify and separate all the resulting intervals. Same goes for the distances as we only need the minimum and maximum distances, that are not effected by loss of precision. To store the indices of the clusters, we know that they will not exceed 2^{16}, so they can be store as half precision values.

Chapter 5

Experimental Setup

This work evaluates the proposed algorithms for image and text data. This chapter describes the used datasets, the evaluation methods and the experimental setups.

5.1 Datasets

5.1.1 Never Ending Language Learner data

The Never Ending Language Learner (NELL) is a learning agent that reads the web, extracts data and verfies beliefs [2][3]. NELL for example knows that "Pittsburgh" is located in "Pennsylvania". These beliefs represent different noun-phrases such as "Pittsburgh" and "Pennsylvania". The noun-phrases belong to certain categories. "Pittsburgh" is a "City" and "Pennsylvania" is a "State". These subcategories both belong to the main category "Geopolitical Location". While there are already different subcategories, the goal for a hierarchical clustering algorithm here is to extract new useful subcategories.

The used dataset, extracted web-information by NELL, contains 32 different main categories, such as "Animal", "Location" or "Person". Each of these consists of up to 250 different entities that belong to different subcategories. Examplary entites for the category "Animal" are "Otter", "Squirrel" or "Wolf".

This thesis shows in chapter 6 the learned subcategories.

5.1.2 MNIST handwritten digits

The MNIST handwritten digit database contains images of the handwritten digits from zero to nine [4]. Samples of these images are shown in figure 5.1 Its training set contains a total of 60,000 images, where each image is represented as a 784-dimensional vector corresponding to a greyscale image with 28x28 pixels.

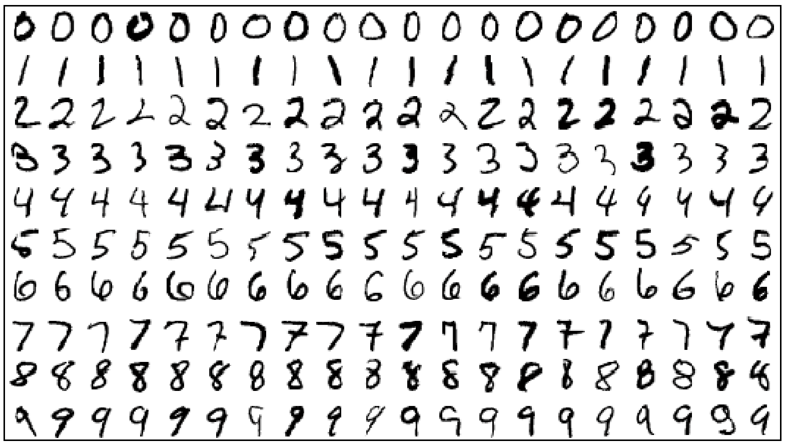

FIGURE 5.1: The MNIST handwritten digits database contains 60,000 greyscale images of handwritten digits ranging from zero to nine. These samples show ten randomly drawn samples for each label represented as a 28x28 pixel image [4].

The goal of clustering MNIST images is to find an unsupervised learning method that can distinguish between greyscale images. In addition, we can define various clustering tasks where we pick a subsample of the ten labels and then try to transfer the results to other subsamples. For example, we first cluster images labeled as zero, one, two, three or four and later apply the knowledge the learned gained for clustering images labeled as five, six, seven, eight or nine. Theses types of experiments allow high-level transfer learning if we define several different clustering tasks, e.g. for five different labels there are $\binom{10}{5} = 252$ different combinations of labels.

Another obsevation that results from hierarchical clustering is the similarity of different labels, i.e. which labels are likely to get clustered together.

5.1.3 CIFAR-10

Another image dataset this thesis uses for evaluation is the CIFAR-10 dataset that contains 60,000 RGB images of ten different categories [5]. Each image consists of 32x32 pixels and is thus represented as a 3072-dimensional vector (32x32x3). The categories and ten random images from each are shown in figure 5.2.

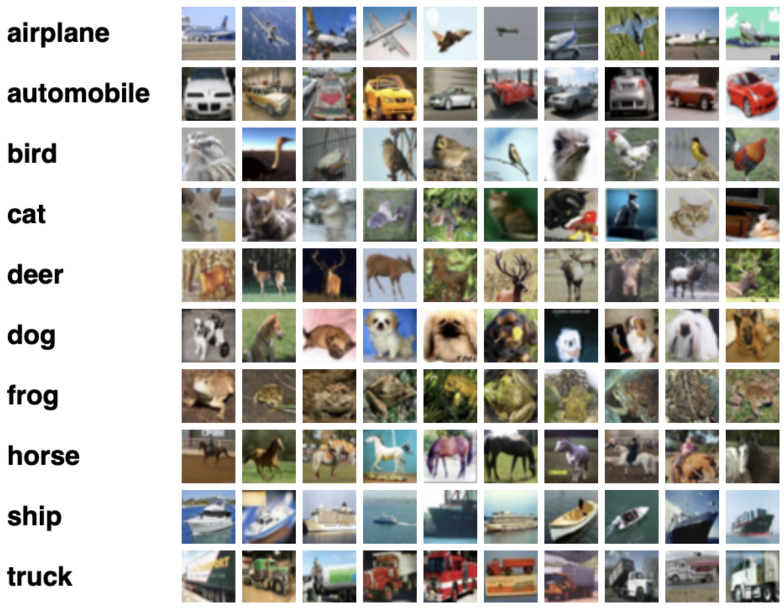

FIGURE 5.2: The CIFAR-10 database contains 60,000 RGB images of the ten shown different classes. These samples show ten randomly drawn samples for each label represented as a 32x32 pixel image [5].

As the amount of images and the amount of classes is equal to the ones in the MNIST database, we can also try similar experiments. The main difference is that the images consist of RGB pixels instead of greyscale values.

5.1.4 CIFAR-100

The CIFAR-100 dataset contains similar images, but instead of 6,000 images each for 10 classes, it consists of 600 images each for 100 classes. The classes are divided into 20 superclasses each containing five subclasses. Examples of superclasses and corresponding subclasses are shown in table 5.1.

superclass	subclasses
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle

TABLE 5.1: The CIFAR-100 dataset contains 20 different superclasses, each with five different subclasses leading to 100 classes overall. The images are represented in the same way as in the CIFAR-10 dataset, i.e. by a 3072-dimensional vector [5].

Having superclasses and subclasses allows clustering between different subclasses within a superclass and also between different superclasses. This allows more experiments than for the CIFAR10 data.

5.1.5 Omniglot

The omniglot dataset contains 1623 handwritten characters from 50 different alphabets, where each character is represented by 20 different images. Each image is grayscale and represented by 105x105 pixels [6]. Figure 5.3 shows characters of the more well-known Latin, Greek and Hebrew alphabets that are part of the dataset.

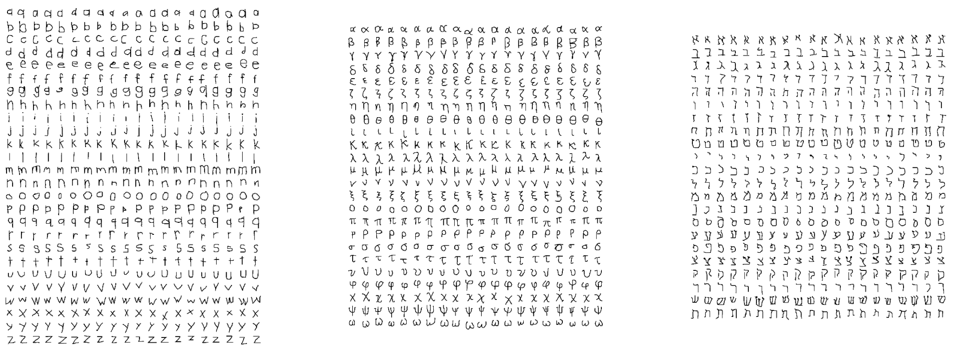

FIGURE 5.3: The omniglot dataset contains handwritten characters of different alphabets, such as Latin, Greek and Hebrew [6].

5.2 Cost functions

In order to evaluate the quality of a clustering, we need some kind of cost function that compares the generated clustering $C_1, ..., C_k$ with the target clustering $C_1^*, ..., C_k^*$. One method to compare them is the majority distance as shown in equation 5.1 where n ist the number of sampled points.

$$cost_{majority}(C_{1:k}, C_{1:k}^*) = \frac{1}{n}\sum_{i=1}^{k}(\|C_i\| - \max_j \|C_i \cap C_j^*\|) \tag{5.1}$$

This cost function is motivated by finding corresponding clusters with the lowest distance, i.e. each generated cluster gets matched with the optimal target cluster. However two generated clusters can be matched with the same target cluster. This motivates the hamming distance as shown in figure 5.2.

$$cost_{hamming}(C_{1:k}, C_{1:k}^*) = \max_\sigma \frac{1}{n}\sum_{i=1}^{k}(\|C_i\| - \max_{\sigma,j} \|C_i \cap C_j^*\|) \tag{5.2}$$

However, the hamming distance consists of an assignment problem to find the optimal matching σ between the generated clusters and the target clusters. Table 5.2 shows how such a matching can look like.

j\i	1	2	3	4	5
1	20	15	30	50	40
2	80	10	15	20	30
3	20	30	50	80	60
4	30	50	40	20	10
5	20	30	40	50	25

TABLE 5.2: In order to calculate the hamming distance between two clusterings, we have to calculate the optimal mapping that results in lowest distance for these two clusterings. For random distances between clusterings $C_1^i, ..., C_k^i$ and $C_1^j, ..., C_k^j$ we can calculate the optimal mapping (blue highlighted cells) in a brute force way or more efficiently with the hungarian method [7][8].

While solving the assignment with a brute force strategy would result in $O(n!)$ complexity, Harold Kuhn introduced the hungarian method to solve the problem in $O(n^4)$ complexity [7]. Later on, James Munkred modified the algorithm to $O(n^3)$ complexity [8]. A detailed explanation of the hungarian method is included in appendix A.

5.3 Experiments

In order to find new subclusters for the NELL data, we cluster each of the 32 main categories seperately. This results in 32 different clustering tasks, where we compare the results of each clustering task with the target labels using the majority distance function. We will receive a cost function $cost(\alpha)$, that shows us for which value of α the resulting clusterings are good, for each category. By averaging all cost functions, we know for which values of α the α-linkage performs well in general. Beside having a value of α that can be used for other clustering tasks, the experiments also give different representation levels of clusters that are discussed in section 6.

To cluster the image data, we set up $\binom{10}{5} = 252$ different experiments by selecting all combinations of five out of the ten labels. In order to do so in efficient time, we subsample the dataset to 60 points for each label, so one experiment will cluster 300 points. By having a fixed set of point, we can show that a certain value of α will lead to good results for the subsampled data. We will use this kind of experiments for all in section 5.1 mentioned image datasets where all RGB-channels are treated equally for colored images.

In addition to these experiments, we will try to cluster as diverse as possible superclasses of the CIFAR100 dataset by manually picking the five superclasses fish, flowers, household furniture, people and vehicles 1. For each superclass we pick one subclass and evaluate the results for all $5* \binom{5}{1} = 25$ different combinations of subclasses. In addition to the experiments with $k = 5$ clusters, we compare these results to the results for picking two different subclasses of each superclass ($5* \binom{5}{2} = 50$ different experiments) resulting in $k = 10$ clusters and also for picking three different subclasses ($5* \binom{5}{3} = 50$ different experiments) resulting in $k = 15$ clusters.

In comparison to picking as diverse as possible superclasses, we also evaluate the performance for as similar as possible subclasses. Similar subclasses are already given in the dataset through the subclasses within one superclass. We then evaluate the majority and the hamming cost for each superclass and again average the cost over all 20 superclasses to evaluate an optimal value for the parameter α.

The results of these experiments are discussed in the following section 6.

Chapter 6

Results and Discussion

6.1 Clustering Text Data

6.2 Clustering Image Data

Chapter 7

Conclusion

Appendix A

The Hungarian Method

Our goal is to find the best possible matching between two clusterings $C_1^i, ..., C_k^i$ and $C_1^j, ..., C_k^j$. In order to do so, we calculate the cost of matching each possible pair of clusters within the two clusterings.

To find the optimal matching in a brute force way, we have to look at each possible matching. Say we want to match each i to one j. For $i = 1$ we can pick from 5 different values of j, for $i = 2$ there are 4 potential values of j. This will overall result in $k! = 5! = 120$ different combinations, thus the complexity of the brute force approach is $O(k!)$. A more efficient algorithm (especially for higher values of k) was introduced by Kuhn and Munkres [7][8]. It consists of three major steps. In the first one, we subtract the row minima from each row. This step is performed in table A.1.

j\i	1	2	3	4	5	
1	5	0	15	35	25	(-15)
2	70	0	5	10	20	(-10)
3	0	10	30	60	40	(-20)
4	20	40	30	10	0	(-10)
5	0	10	20	30	5	(-20)

Table A.1: Hungarian method step 1: Subtract the row minima from each row.

After subtracting the row minima, we now also subtract the column minima from each column as shown in table A.2.

Now we try to find the optimal matching. To do so, we cover all zeros with lines and count the minumum needed lines to do so. Table A.3 shows that we need four lines.

After covering the zeros and counting the lines, we found the optimal matching in case the number of lines equals the number of rows (or columns) in the matrix. As we need four lines and the matrix has five rows in this example, we have to add more zeros.

j\i	1	2	3	4	5
1	5	0	10	25	25
2	70	0	0	0	20
3	0	10	25	50	40
4	20	40	25	0	0
5	0	10	15	20	5
	-	-	(-5)	(-10)	-

Table A.2: Hungarian method step 2: Subtract the column minima from each column.

j\i	1	2	3	4	5
1	5	0	10	25	25
2	70	0	0	0	20
3	0	10	25	50	40
4	20	40	25	0	0
5	0	10	15	20	5

Table A.3: Hungarian method step 3: Cover all zeros with as few lines as possible.

To do that, we subtract the minimum value of the matrix (which is 5 here) from all uncovered values that are not zero and add it to all values that are not zero and covered twice. Now we can again check the needed lines as in table A.4.

j\i	1	2	3	4	5
1	5	0	5	20	20
2	75	0	0	0	20
3	0	10	20	45	35
4	25	45	25	0	0
5	0	10	10	15	0

j\i	1	2	3	4	5
1	5	0	5	20	20
2	75	0	0	0	20
3	0	10	20	45	35
4	25	45	25	0	0
5	0	10	10	15	0

Table A.4: Hungarian method additional step: Create more zeroes until the number of minimal needed lines to cover all zeros matches the number of rows.

This will then result in the assignment seen in table A.5. Applying the matching to the input matrix then gives the optimal cost by summing the optimal values. For this example the optimal cost is then 95.

j\i	1	2	3	4	5
1	5	0	5	20	20
2	75	0	0	0	20
3	0	10	20	45	35
4	25	45	25	0	0
5	0	10	10	15	0

j\i	1	2	3	4	5
1	20	15	30	50	40
2	80	10	15	20	30
3	20	30	50	80	60
4	30	50	40	20	10
5	20	30	40	50	25

Table A.5: Result of the hungarian method: The optimal matching between two clusterings.

Bibliography

[1] Maria-Florina Balcan, Vaishnavh Nagarajan, Ellen Vitercik, and Colin White. Learning-theoretic foundations of algorithm configuration for combinatorial partitioning problems. *CoRR*, abs/1611.04535, 2016. URL http://arxiv.org/abs/1611.04535.

[2] T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner, B. Kisiel, J. Krishnamurthy, N. Lao, K. Mazaitis, T. Mohamed, N. Nakashole, E. Platanios, A. Ritter, M. Samadi, B. Settles, R. Wang, D. Wijaya, A. Gupta, X. Chen, A. Saparov, M. Greaves, and J. Welling. Never-ending learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 2302–2310. AAAI Press, 2015. ISBN 0-262-51129-0. URL http://dl.acm.org/citation.cfm?id=2886521.2886641.

[3] T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, B. Yang, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner, B. Kisiel, J. Krishnamurthy, N. Lao, K. Mazaitis, T. Mohamed, N. Nakashole, E. Platanios, A. Ritter, M. Samadi, B. Settles, R. Wang, D. Wijaya, A. Gupta, X. Chen, A. Saparov, M. Greaves, and J. Welling. Never-ending learning. *Commun. ACM*, 61(5):103–115, April 2018. ISSN 0001-0782. doi: 10.1145/3191513. URL http://doi.acm.org/10.1145/3191513.

[4] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

[5] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[6] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015. ISSN 0036-8075. doi: 10.1126/science.aab3050. URL http://science.sciencemag.org/content/350/6266/1332.

[7] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[8] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.