Handwritten Digit Recognition using Machine Learning, Deep Learning and Metric Learning Approaches

Allwyn Joseph [*1] **Nisal Upendra** [*2] **Rohil Gupta** [*3]

Abstract

In this work, we focus on the classical machine learning problem of digit recognition using state-of-the-art and classical machine learning approaches. We effectuate this using two primary methods: Structural method using Freeman Codes to capture explicit digit features and Numerical method using Deep Learning to capture implicit digit features. These methods were used in combination with the classical lazy Nearest Neighbours algorithms to predict digits from the learnt features. Additionally, sequence mining algorithms were used to visualize the patterns/features captured by the Freeman Codes used with the structural method. Finally, a GUI (found in this link) was implemented in an effort to demonstrate the efficacy of our models and approach towards recognizing digits. The code and related literature for our work can be found in this link.

1. Introduction

This project deals with the hand written digits to recognize them using Machine Learning techniques. The project is sub-divided into three major parts named as Structured Method, Numerical Methods and Sequence Mining.

1. **Structured Method:** This method deals with the Freeman Code representation of the digits in the images. The name structured arises from the fact that the method deals capturing the visual/physical structure of a given input data. In our case the input data are digit images and the structure of digit images are extracted using Freeeman's-code (explained in further detail in section 4. These extracted image structures are then used to predict foreign digit images by means of a modified version of the edit-distance algorithm. Section 4 will dive deeper into the experimentation methodology, challenges confronted and results obtained employing the Structured Method for digit recognition.

2. **Numerical Method**: The name 'numerical' comes from the fact that this method solely deals with a numerical representation of the input data. In essence the method entails using feature embeddings extracted from digit images (using deep learning approaches) to predict new external digit image examples. At first, the predictions are realized using the standard lazy KNN algorithm and later it is used in combination with the LMNN metric learning algorithm to realize more accurate predictions. In section 5 we will dwell further into the experimentation route and the respective results obtained using aforementioned methods.

3. **Sequence Mining**: The Sequence mining exercise in this project is concerned with using frequent sequential mining algor hms to mine frequent patterns in the freeman codes of the digits dataset used. We have also attempted to use these mined patterns to recognize handwritten digits by comparing the freeman code of the input digit with pre mined freeman code patterns. We have used the Prefixspan algorithm using PMF (an open-source data mining mining library) to generate patterns. In 6, we will discuss further into the algorithm used and our methods to compare and identify handwrtitten digit input using freeman codes.

2. Algorithms

2.1. K Nearest Neighbours (KNN)

The K-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression. K-NN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification. The k-NN algorithm is among the simplest of all machine learning algorithms. Algorithm 1 explains how to classify an unknown example

[1]University Jean Monnet, Saint-Etienne, France [2]University Jean Monnet, Saint-Etienne, France [3]University Jean Monnet, Saint-Etienne, France. Correspondence to: <allwyn.joseph@etu.univ-st-etienne.fr>, <nisal.upendra@etu.univ-st-etienne.fr>, <rohil.gupta@etu.univ-st-etienne.fr>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

x using a dataset *S*. In the scope of the project, the distance used is either Euclidean Distance for Numerical method and Edit Distance for Structural method.

Algorithm 1 K-Nearest Neighbours

Input: **x**, *S*, *d*
Output: class of **x**
for (x',*l'*) $\in$ *S* **do**
  Compute the distance d(x',x)
end for
Sort the |S| distances by increasing order.
Count the number of occurrences of each class l_j among the *k* nearest neighbors.
Assign to x the most frequent class.

2.2. Condensed Nearest Neighbours (CNN)

To be able to identify the digit and reduce the number of distance to be calculated there is an optimization needed by reducing the size of the database to reduce the computations. Two algorithms are proposed to speed up the classification process: one in Algorithm 2) which removes from the original dataset S the outliers and the examples located in the Bayesian error region, smoothing the decision boundaries, and other one in Algorithm 3 which deletes the irrelevant examples (e.g. the centers of the homogeneous clusters)

Algorithm 2 Step1 : Remove from the original dataset S the outliers and the examples of the Bayesian error region

Input: *S*
Output: S_{cleaned}
Split randomly *S* into two subsets S_1 and S_2
repeat
  Classify S_1 with S_2 using the 1-NN rule
  Remove from S_1 the misclassified instances
  Classify S_2 with the new set S_1 using the 1-NN rule
  Remove from S_2 the misclassified instances
until stabilization of S_1 and S_2
$S_{\text{cleaned}} = S_1 \cup S_2$

2.3. Deep Convolutional Denoising Auto-encoders (DC-DAE)

An auto-encoder(AE) is a network that is a result of stacking together two multi-layer perceptron (MLP). The first MLP acts as an encoder (encoding the input to the MLP) and the second one as a decoder (decoding the output of the encoder MLP). As the network is trained in a unsupervised manner, it seeks to learn a representation (encoding) of the input data in an effort to reduce its dimension, while simultaneously capturing salient features.

Within the scope of our work, the algorithm served to create embeddings and generate new digit images, as will be described in further detail in section 3)

Algorithm 3 Step2 : Remove the irrelevant examples

Input: *S*
Output: STORAGE
STORAGE $\longleftarrow \emptyset$; DUSTBIN $\longleftarrow \emptyset$
Draw randomly a training example from *S* and put it in STORAGE
repeat
  for $x_i \in S$ **do**
    if x_i is correctly classified with STORAGE using the 1-NN rule **then**
      STORAGE DUSTBIN $\longleftarrow x_i$
      STORAGE $\longleftarrow x_i$
    end if
  end for
  STORAGE $\longleftarrow$ STORAGE $\setminus$ DUSTBIN
until stabilization of STORAGE
Return STORAGE

Algorithm 4 DC-DAE Algorithm

for number of training iterations **do**
  Sample $x^{(i)} \in \{x^{(1)}, ..., x^{(n)}\}$ of size m
  $a = x^{(i)}$ + random stochastic noise
  for l layers **do**
    $z^{[l]} = w^{[l]} * a + b^{[l]}$
    $a = g(z^{[l]})(g : linear/non-linear function)$
  end for
  $\hat{x}^{(i)} = a$
  loss = $\sum_{i=1}^{m} \|x^{(i)} - \hat{x}^{(i)}\|$
  Back-propagate, update w and b for all layers
end for

2.4. Deep Convolutional Generative Adverserial Networks (DCGAN)(Radford et al., 2015)

The DCGAN algorithm is a direct inspiration from the GAN algorithm developed by Ian Goodfellow (Goodfellow et al., 2014). The algorithm involves two model, a generative model G and a discriminative model D. Model G tries and captures the distribution of a training dataset in an effort to emulate and generate data based on its distribution. Model D tries to discriminate as to whether the presented sample comes from the training data or G. This framework constitutes a two-player min-max game where G tries to trick D into predicting that the sampled data is from the training set, while D tries to avoid just this. In theory, when the models are done training, everytime data is sampled (from the training set or G), D predicts with an equal probability of 0.5 that the data is from the training set or G.

Algorithm 5 GAN Algorithm

for number of training iterations **do**
  for k steps **do**
    • Sample mini-batch of m noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from noise prior $p_g(z)$
    • Sample mini-batch of m noise examples $\{x^{(1)}, ..., x^{(m)}\}$ from train data distribution $p_{data}(x)$
    • Update the discriminator by ascending its objective function
    $\frac{\partial}{\partial\theta_d}[\frac{1}{m}\sum_{i=1}^{m}[logD(x^{(i)}) + log(1 - D(G(z^{(i)})))]]$
  end for
  • Sample mini-batch of m noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from noise prior $p_g(z)$
  • Update the discriminator by descending its objective function
  $\frac{\partial}{\partial\theta_d}[\frac{1}{m}\sum_{i=1}^{m} log(1 - D(G(z^{(i)})))]$
end for

The pseudo-code 5 shows how the GAN is trained. The DCGAN is simply a deep convolutional version of the same. The algorithm comes into good use for dataset generation (expounded on in section 3.1)

2.5. Large Margin Nearest Neighbor (LMNN) - Metric Learning (Weinberger & Saul, 2009)

The LMNN algorithm is based on the idea of metric learning wherein a metric is learnt such that it assigns small distances to pairs of similar examples and large distance to dissimilar examples. The LMNN algorithm is a subset of metric learning algorithms, specially tailored for use with KNN. The idea is based on two sets of constraints, Similar and Relative on two categories of examples: "Target neighbour" and "Imposters".

$S = \{(x_i, x_j) : y_i = y_j \quad and \quad x_j \text{belongs K-n of} x_i\}$,

R = $\{x_i, x_j, x_k) : (x_i, x_j) \in S, y_i \neq y_k\}$

The equation above simply suggests that elements that belong to the same class be put in one set S (similar) thereby bringing them closer. And, increase distance between pairs of elements in set S and elements of opposite class. The soft formulation of the LMNN algorithm is a follows:

$$\min_{M\succeq 0}(1-\mu)\sum_{(x_i,x_j)\in S} d_M^2(x_i,x_j) + \mu\sum_{i,j,k}\xi_{ijk}$$

s.t $\quad d_M^2(x_i,x_k) - d_M^2(x_i,x_j) \geq 1 - \xi_{ijk} \quad \forall(x_i,x_j,x_k) \in R$,

where μ controls the push/pull trade-off

2.6. PrefixSpan Algorithm

This sequential pattern mining algorithm finds statistically relevant patterns between data examples where the values are delivered in a sequence. PrefixSpan mines the complete set of patterns but greatly reduces the efforts of candidate subsequence generation by recursively projecting prefixes into smaller projected databases and by growing subsequence fragments in each projected database.(Pei et al., 2004)

Algorithm 6 PrefixSpan

Input: A sequence database **S**,
minimum support threshold **ms**
Output: The complete set of sequential patterns **S'**
Subroutine: *PrefixSpan*(α, L, $S|\alpha$)
Parameters: α: sequential pattern
L: the length of α
S|α: : the α-projected database, if $\alpha \neq <>$
the sequence database **S**
Recursively call: PrefixSpan(<>,0,**S**).

3. Dataset and Embeddings

The data to build our digit recognition system came primarily from the MNIST dataset (LeCun & Cortes, 2010). The dataset was further augmented using the DC-DAE and DCGAN algorithms (explained in section 3.1). From an amalgam of this data the embeddings were then generated using only the encoder part of the DC-DAE trained on the MNIST dataset.

3.1. Dataset Generation

As mentioned above and can be seen from figure 1, the final dataset is an amalgam of data from MNIST, generated data from the DC-DAE and generated data from the DCGAN.

Firstly, both the DC-DAE and the DCGAN were trained on the complete MNIST dataset. Ten separate DCGANs are trained on datasets of individual digits from 0 to 9, to be able to generate specific digits after the networks have been trained. Once the training was terminated, both the networks had learnt weights that captured structural and non-linear patterns from the digits images. Further on, the trained DC-DAE was fed with 25,000 randomly picked digit images from the MNIST database to generate 22,500 images. This was followed by feeding each of the ten trained DCGANs with random noise to generate 1,250 images each, resulting in a total of 12,500 images. 22,500 digit images from DC-DAE was combined with 12,500 digit images from DCGANs and 35,000 randomly picked digit images from the MNIST database to create the final dataset of 70,000 digit images (60,000 train images and

10,000 test images)

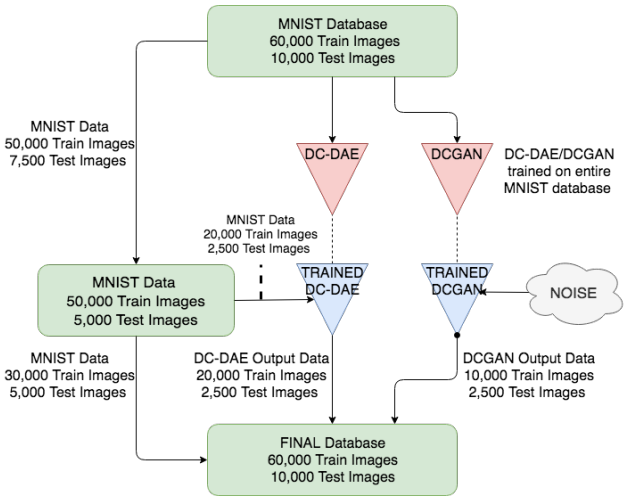

Figure 1. Database construction flowchart

3.2. Embedding Generation

Generation of embeddings served two purposes, furthering experimentation using numerical methods (discussed in section 5) and selection of strong digit examples (using the CNN algorithm) from the final dataset so as to reduce computational of running algorithms associated with structural and numerical methods.

In essence, the final dataset was fed to the DC-DAE and the output was recovered from the encoder part of the network, resulting in a vector of lenght 197 (196 feature values and 1 label value). Further on, from the generated 70,000 digit embeddings, about 5,000 strong embeddings were selected using the CNN algorithm (on which a majority of the experimentation for Structural and Numnerical methods were conducted).

Note : Concerning the numerical method for digit recognition on the online platform (described in section 7), only the MNIST database was utilized. This was because the digits generated on the platform were represented in a binary format, without blemishes or weak pixelation common in hand-written digit image representations. To ensure robust performance of the numerical method on the platform generated digit representations, the normalized MNIST data was was converted to its respective binary representation. An auto-encoder model was trained using the binary MNIST dataset as input, which was eventually used to generated embeddings for the dataset. On these embeddings, the CNN (section 2.2) algorithm was applied to extract only strong instances. Further on, a metric is learnt on the remaining embeddings using LMNN (section 2.5). Finally, during test time, the learnt metric is used in combination with KNN on the strong embeddings to predict the test instance.

4. Structural Method

This method involves getting an image from the platform,converting it to its corresponding freeman-code (figure 2) and computing the edit distances between the generated sequence and freeman-code training sequences. Following this, we get the majority vote from it's K-Nearest neighbours in terms of cost to finally decide the label of the written digit. There are many fine details involved in this method and it is vividly discussed in Methodology Adopted Section.

The main challenge involved in this method was to reduce the computation time involved in predicting the label of the digit. This challenge is addressed properly throughout the implementation of this project.

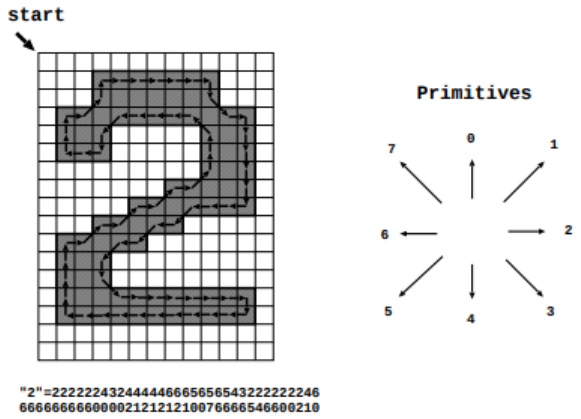

Figure 2. Representation of the digit 2 with noise elements

4.1. Methodology and Challenges

Various challenges were faced during the course of our experiments, details on it and work-arounds adopted are explained in following sections.

4.1.1. Freeman Code Generator

The Freeman Chain Code generator follows the algorithm precised in the given link. The algorithm finds a starting pixel from top left and then (with the help of an inbuilt compass) follows the outer contour of the digit until it reaches the initial starting pixel. In this process, at every step (pixel) a direction is assigned based on the algorithm's trajectory along the contour.

Challenge Involved: The Freeman Code generator algorithm was not robust to noise or unwanted pixels in the input image. This was a problem because, were there to be unwanted pixels, the algorithm would enventually find those pixels, create the freeman chain code for that noise and end the algorithm execution.

Figure 3. Representation of the digit 2 with noise elements

Figure 3 clearly illustrates this phenomenon, where the algorithm generates freeman-code for the upper left noise blob and terminate execution.

Solution: This problem was addressed with the help of a heuristic which involved regenerative freeman chain code. This means the algorithm will iteratively produce the freeman chain code for the continuous set of pixels by removing the particular set of pixel at each iteration until a blank image is obtained. The assumption involved in this is that, the digit will have the longest freeman chain code and therefore is selected as the freeman chain code for the image. This method produced robust results.

4.1.2. Modification of Minimum Edit Distance Algorithm

The Classic version of Minimum Edit Distance algorithm gives the Levenshtein distance (or Edit Distance) between two strings **x** = $x_1.....x_T$ and **y**= $y_1.....y_V$ is given by the minimum number of edit operations needed to transform **x** into **y**, where an operation is an insertion, deletion, or substitution of a single character.

Challenge Involved: The classic version assigns same edit cost to the transformation that involves the substitution of 0 to 7 compared to 0 to 1. Distance computed in this manner will not follow the normal axioms of a true distance metric, i.e transitivity or triangle inequality. It can be seen in the figure 4 also that the transformation from 0 to 7 involves 7 edit operations to turn the direction of from 0 to 7. If this is compared to 0 to 2 transformation, than it involves 2 edit operations thus the 2 times the minimum edit cost is to assigned.

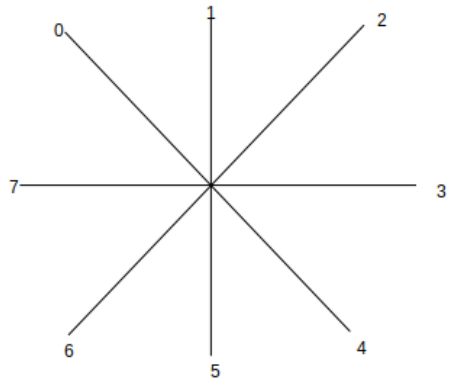

Figure 4. Compass of the Freeman Code Generator Algorithm

Solution: This problem was resolved by assigning different cost to different edit operations, so if it involved tuning the compass once, it was given a minimum cost of 1/7, turning the compass twice was given double the cost of the minimum cost and so on. This resulted in an true distance metric.

4.1.3. Speeding up the K-Nearest Neighbours

K-Nearest Neighbour algorithm (as discussed in sections 1) is a lazy algorithm, that means it does not learn an hypothesis to predict the class labels. To predict the label of a given test instances, it computes the distances between the instance and all the samples in the training set and picks its label based on the labels of the first k samples it is closest to.

Challenge Involved: The issue with this implementation is the computation complexity in producing results. Due to high computation complexity of the modified classic minimum edit distance, realizing a single prediction will takes about 50 seconds. This computation time is not feasible in any real implementation. So, there is a need to speed up the K-Nearest Neighbours algroithm.

Solution: The solution to this problem is addressed with following steps:

- The first step involved the implementation of the Condensed Nearest Neighbours algorithm given in 2 and 3. This reduced the number of training examples from 60,000 images to 5065 images. The remaining images were representation of the numbers, thus could be used to implementing the KNN algorithm.
- The second step has us implement another optimiza-

tion technique to speed up the original algorithm by speeding up the search for the nearest neighbor using the triangle inequality property of the distance function (Aibar et al., 1995). The algorithm was further improved which can be seen in algorithm 7 and 8.

This algorithm 7 takes as an input **Matrix of Edit Distances**. Each row of this matrix has an image's freeman code compared with all the other image's in the dataset in terms of edit distance and the value is stored in the corresponding cell. This matrix is N*N shape with N representing the total number of sample image's freeman-codes.

This algorithm 8 takes as an input **Average Edit Distance per class Matrix**. This matrix is a 10*10 matrix with each row has the average edit distance of the label with other labels. For example, the first row has the average edit distance of 0 class with all the other classes from 1 to 9. The abs(*Cost1-Cost2*) in algorithm 7 represents the smaller circle centered at z in figure 5 and abs(*Cost1 + Cost2*) represents the outer circle centered at z in figure 5.

Algorithm 7 Speeding Up the KNN with implementing triangle inequality

Input: Test Instance, Training Samples, Matrix of Edit Distances, Average Edit Distance per class Matrix
Output: Reduced Training Samples
Two optimally selected examples = Select 2 optimal examples from the training samples with algorithm 8
Compute the corresponding edit cost of test instance with two examples selected and obtain *Cost1* and *Cost2*.
Initialize: *Index of the larger cost example*
if *Cost1* $>$ *Cost2* **then**
  Index of the larger cost = Index of the first example
else
  Index of the larger cost = Index of the second example
end if
From the Matrix of Edit Distances get the array corresponding to the *Index of the larger cost*.
for $x_i \in$ Array of Edit Distances for the *Index of the larger cost* **do**
  if $x_i <$ abs(*Cost1-Cost2*) **or** $x_i >$ abs(*Cost1 + Cost2*) **then**
    Remove the corresponding example from the training sample with edit distance equal to x_i.
  end if
end for
Return Reduced Training Samples

Algorithm 8 Selecting the two most optimal examples for comparison

Input: Test Instance, **Training Samples**, Average Edit Distance per class Matrix
Output: Two optimally selected examples
L_t = *Length of Test Instance's Freeman Code*
New DataFrame for First Example selection = Select all the examples with freeman code length less than L_t - 1 and greater than L_t + 1 .
for $x_i \in$ *New DataFrame for First Example selection* **do**
  Compute the edit distance cost of x_i with test instance and save that value in the corresponding index of the dataframe.
end for
First Example = Select the example with least cost with the test instance.
L_1 = Label of the **First Example**
L_2 = Label of that class which has the closest cost with the label of the **First Example** in in Average Edit Distance per class Matrix.
L_3 = Label of that class which is at the maximum cost with the L_2 in Average Edit Distance per class Matrix.
New DataFrame for Second Example selection = Select all the examples from the **Training Samples** with label L_3.
for $x_i \in$ *New DataFrame for Second Example selection* **do**
  Compute the edit distance cost of x_i with test instance and save that value in the corresponding index of the dataframe.
end for
Second Example = Select that example from the *New DataFrame for Second Example selection* that has the maximum cost.
Two optimally selected examples = **First Example** $\cup$ **Second Example**
Return: Two optimally selected examples

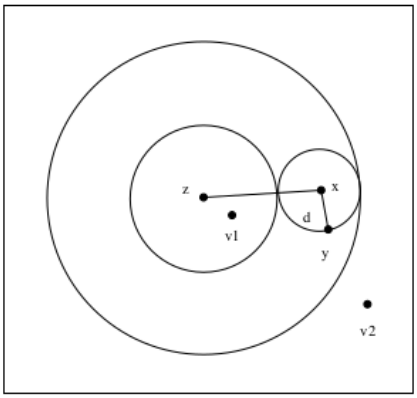

Figure 5. Triangle Inequality used in algorithm 7

After the implementation of the above mentioned algorithms, a significant improvement in computation time was observed, the resits of which are discussed in further detail in the next section.

4.2. Results

The results obtained for the Structural method are discussed in this section.

The computation time for the Structural method reduced to a satisfactory level. Additionally, the performance was not compromised after having applied the optimization algorithms. Figure 6 plots the different values of K-Nearest Neighbours and the accuracy obtained by the Structural method on test dataset for each of the k values.. The accuracy for k = 50 was found to be 0.90 on the test-set.

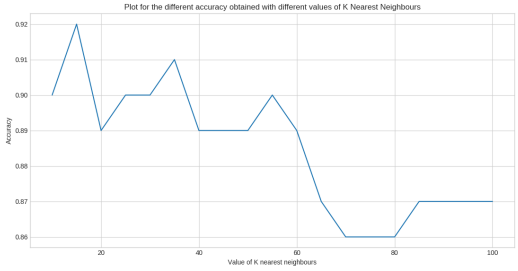

Figure 6. Plot for the different accuracy obtained with different values of K Nearest Neighbours

The results from the GUI platform were also robust to noises and different ways in which a digit can be written.

5. Numerical Method

As opposed to the work on structural representation of digit images (using the Freeman's-code) in the previous section, experiments here were conducted on numerical feature vectors or embeddings of digit images.

As mentioned earlier, the final dataset of digit images were transformed into feature vectors or embeddings of length 196 using a DC-DAE network 2.3 trained on the MNIST database. In theory, the produced embeddings are robust representations of image digits due to the innate ability of neural nets to capture the underlying non-linear structures of input data. Once the embeddings were generated, they were randomly split into train and test sets (With the same train:test ratio as with the MNIST dataset) on which various experiments were conducted. The following sections will detail the methodology and results of each experiment.

5.1. Experiment Two - KNN on All and Strong Instances

Note : The accuracy value at which k was computed is $k = sqrt(\text{number of training instance / } 2)$

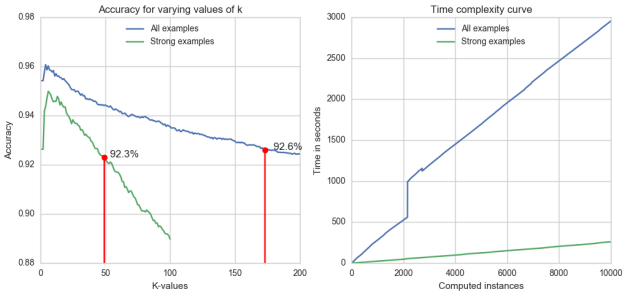

Figure 7. Accuracy and Time complexity curves on running KNN on All and only Strong instances

Experiment One was straightforward and involved two steps:

- First, predict the labels of the test embeddings using KNN with all of the train embeddings (60,000 instances)
- Second, apply the CNN algorithm explained in section 2.2 on the train embeddings to recover only relevant instance (4707 instances). Following this, predict the labels of the test embeddings using KNN with only the relevant instances. The relevant instances are also regarded as strong instances as they form the backbone of the data distribution.

Figure 7 plots the results obtained following the above steps. It is clear from the plots that preserving relevant examples while discarding the irrelevant ones, not only retained the accuracy scores (92% at k = 49), but also sped up the computation 10 folds.

5.2. Experiment Two - KNN + LMNN on Strong Instances

Having condensed the train embeddings to only about 7% of its original size, we then wanted to verify as whether the accuracy scores could be ameliorated. To achieve this we resorted to the LMNN algorithm described in section 2.5.

The main intuition behind LMNN was to learn a pseudo-metric under which instances in the training set are surrounded by at least k instances with the same label. To this end, a metric-k-value had to be defined to successfully run this algorithm. So k-values ranging from 10 to 70 with a step size of 10 were considered while the algorithm was fit onto the training data.

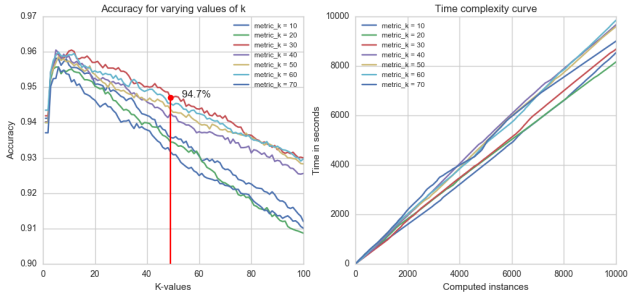

Figure 8. Accuracy and Time complexity curves on running KNN with LMNN on only Strong instances

Figure 8 illustrates the results obtained after running the KNN algorithm along with LMNN. It is clear from the plots that, for a metric-k-value of 30 (at k = 49) the highest accuracy of 94.7% is achieved. This is a two point improvement over the vanilla KNN algorithm. This being said, the time complexity curves show that, the combination takes a toll on the computation time; 10 times slower than the vanilla version. This deceleration could be attributed to the matrix multiplication step introduced to calculate the new distances between a pair of points.

5.3. Experiment Summary

Table 1 recapitulates the results of the conducted experiments. The "Max Accuracy" column projects the highest values of accuracies (calculated by applying KNN on the test embeddings) at the respective k-values. Since this accuracy fluctuated with different distributions, we went with calculating the accuracy at a more robust value for k (formula in section 5.1). Column "Accuracy", projects the accuracy values (calculated by applying KNN on the test embeddings) at robust k-values. Additionally, the metric learnt using LMNN on the train embeddings was with a metric k value of 30.

Train embeddings	Max Accuracy	Accuracy
All	0.961 @k = 3	0.926 @k = 173
Strong	0.95 @k = 5	0.923 @k = 49
Strong + LMNN	0.961 @k = 10	0.947 @k = 49

Table 1. Experiment Summary

6. Sequence Mining

In this part of our work we aim to use Frequent Sequential Pattern mining using the 8-connectivity freeman codes generated from the handwritten digits. For this, we used the SPMF, an open-source java library which has an implementation of the PrefixSpan algorithm, which was discussed earlier in the Algorithms section.

6.1. Pattern Generation from Freeman Code

Firstly, we had to generate freeman codes of the frequent patterns for all the handwritten digits we had collected. Since we already had the freeman code that was used for the Structural method, we classified all these freeman codes by digit and then formatted these freeman codes to suit the required input format for SPMF

6.2. Data Pre-processing and Pattern Mining Process

SPMF accepts strings that have a ending of a sub-sequence specified by a *-1* an the ending of a sequence specified by *-2*. Since there were no sub-sequences related to the freeman code, for a example freeman code of *1,2,3,4,5* the required input pattern would be *1,2,3,4,5,-1,-2*. For the mining we used the minsup value at 50%, and the details regarding this will be discussed in the next chapter.
The pattern mining process limits the maximum length of the patterns to 15. There are two main reasons behind this.

- Avoid unnecessarily longer patterns that would overfit the recognition process
- Reduce similar patterns that are very long but are different only by two or three digits.
- Reduce the time taken to mine patterns
- Uniform length patterns make it easier to remove similar patterns by a voting system based on grouped majority, which will be explained later in this section.

The mined patterns are then grouped by the first three digits in the freeman code. Each group of patterns is then reduced to one pattern by replacing different digits with the most frequent digit at that position. This is made possible by having the same length for all the mined patterns, by removing any patterns that have a length of lesser than 15. Since we do not mine any patterns that are longer than 15, this leaves us with patterns that have a length of exactly 15. These processed patterns are then saved in text format for digit recognition and visualization purposes.

6.3. Experimenting with Minsup Values

An issue we encountered during the mining process was that for lower minsup values, patterns took a long time to mine. Since most freeman codes were between 50 to 70 in length, it would often cause memory issues during the mining process. This also led to having very large ($>$ 100MB with more than a 1,000,000 patterns mined) pattern files with lots of similar patterns. Since we are only concerned with patterns that are common to a majority of the dataset,

we decided to increase the minsup value to 50%, so that only patterns that were present in more than half the dataset would be mined. This helped us not only to reduce the time taken for mining, but also to reduce the time taken for the pattern mining process.

6.4. Results

After the pre-processing, we had a much lower number of patterns and these were much more coherently different than the original set of mined patterns.
For example, the number of mined patterns before and after the pre-processing for the digit 9 is shown below (Terminal output from our code).

```
Nb before removing >15: 76786
Nb after removing <15: 3933
Nb after grouping: 15
```

As shown, we managed to remove the number of patterns to a level where it would be possible to compare with a freeman code input, and then visualize in real time.

6.5. Recognizing Digits using Generated Patterns

The main objective of the sequence mining process was to mine, preprocess and visualize patterns from freeman codes. After this, we explored the possibility of matching the mined patterns with freeman code for a handwritten digit to predict that number. To do this we explored several possibilities of matching.

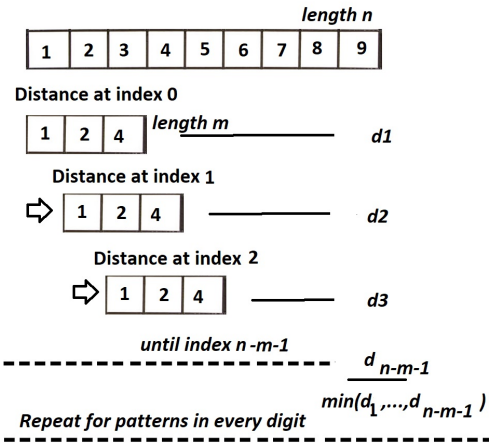

Figure 9. Obtaining minimum distance for a mined pattern against an input freeman code

- Hamming Distance
- Freeman Corrected Hamming Distance

In both these approaches, we used a slice of exactly the same length (15) as the final mined patterns, starting from the first digit in the freeman code and then moving along incrementing by a single position at a time.
We would get the least distance we obtained for a pattern at any position in the input freeman code. After repeating this process, we could get the total of minimum distances for every mind pattern against the input freeman code. Since different digits have a different number of mined patterns, we would then normalize this score by dividing this by the number of scores. After comparing with all the patterns in every digit, we would return the digit where the patterns had the lowest difference per pattern. This process is explained further in the following image.

By using this method, we obtain a minimum distance $d_{min,i}$ for every mined pattern i for a single digit. For a single digit j from 0 to 9, we get the total distance $D_{total,j}$, which we normalize by dividing by the number of patterns k. We then obtain the prediction P by returning the digit j with the lowest distance per pattern.

$$P = min\left(\frac{D_{total,j}}{k}\right)_{j=0}^{9}$$

As mentioned earlier, we used two methods, to calculate the distance between a pattern and a input freeman code segment of the same length. However, using the hamming distance does not provide accurate results as the difference between 8-connectivity freeman digits is not equivalent to the numerical distance. The following image explains this problem for a pattern element of 0. As seen, the distance to another element is not equal to the numerical value.

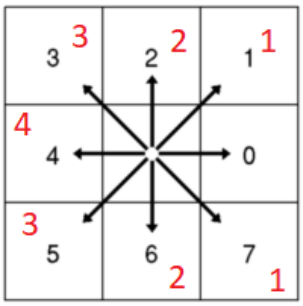

Figure 10. Actual distance for freeman element 0 to other elements

To calculate this, we modified the hamming distance to make a correction for this problem.

```
def _corrected_hamming_distance(a, b):
difference = abs(a-b)
if difference > 4:
```

```
        return 8 - difference
    else:
        return difference
```

Since we had a low number of patterns, it was possible to make a prediction using this method fast enough to incorporate into our GUI. However, when tested against a test sample of freeman codes for every digit from 0 - 9, the accuracy level achieved was very low. The highest achieved was 52.6% digit 4, and 55.4% for digit 5.

6.6. Summary

While, we were able to achieve meaningful patterns filtered out by a large number of generated patterns and managed to visualize the mined patterns, the digit recognition from the mined patterns was not successful.

7. GUI

For the creation of the GUI, we used the following technologies.

- HTML5, Bootstrap *(for canvas and interface)*
- CSS *(for styling and canvas effects)*
- Javascript, JQuery and JSON *(for interface actions, and to handle communications with the server)*
- Python Flask *(to serve as the connection API between the python code and the interface)*

7.1. Image Matrix Generation from HTML Canvas

For this purpose, we have a drawing canvas of 500x500. After a digit has been handwritten to the canvas, we project this image to another hidden canvas of size 28x28. However, we add a margin of 1 pixel to this image to avoid errors caused by the drawn digit overlapping the margin of the original canvas. After this, we convert the data in this hidden canvas to a matrix using the *getImageData()* method in Javascript. This data is then converted into binary image data inside javascript, and then encoded in json and sent as input to the server.

7.2. Freeman Code Visualization

For this part, the objective was to visualize the patterns mined from our images. For this, we drew the image into a numpy array of zeros, by traversing through the matrix according the 8-connectivity freeman elements. We would then convert the image matrix into an image using the *imshow* library in python.

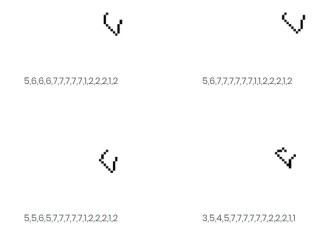

Figure 11. Example mined patterns for 2 from PrefixSpan

8. Reflections and Future Work

8.1. Improving Digit Recognition Accuracy in Structural Methods

A suggestion to improve the accuracy would be to handpick the prominent best features for every digit and use these for the pattern matching. We also hope to look into the possibility of coding our own implementation of PrefixSpan, and to improve the pre-processing to further filter out irrelevant patterns.
We also need to find an alternative than using the modified Hamming distance to find the relation between input freeman code and mined patterns
Another suggested work is to test other algorithms for sequence mining as the patterns generated by PrefixSpan are generally not longer than 20 characters, which for some digits is not enough to generate meaningful patterns.

8.2. Improving the computation time in Structural Methods

The number of distances to be calculated to perform KNN involves almost 5065 training samples. There were techniques used to reduce the number of distances to be calculated. These techniques are explained in algorithm 7 and 8. Particularly in algorithm 8 there is a part involved in selecting the most optimal first example based on the length of the test instance. This takes into account all the examples of training samples with freeman chain code length equal to +1 and -1. This idea works in most cases because mostly the example with same label were selected but it was seen during the testing in some cases the the examples with different labels and almost same freeman chain code length were selected, thus inducing a sub-optimal selection of first example hence not being able to reduce the computation time.

8.3. Improving Digit Recognition Accuracy in Numerical Methods

The numerical methods are only as good as the data at hand. The dataset we put together was primarily based on the MNIST dataset, bringing to the table only a very specific

distribution. However, to offset this, auto-encoders and DC-GAN were used. This being said, very little hyper-parameter tuning was effectuated on these networks to realize optimal results. So a possible improvement could come from network hyper-parameter tuning so as to obtain a more robust dataset and consecutively a better accuracy score. Additionally, other metric learning techniques could be tuned and tested for result amelioration.

References

Aibar, Pablo, Juan, Alfons, and Vidal, Enrique. Extensions to the aesa for finding k-nearest-neighbours. In Ayuso, Antonio J. Rubio and Soler, Juan M. López (eds.), *Speech Recognition and Coding*, pp. 92–95, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-642-57745-1.

Goodfellow, Ian, Pouget-Abadie, Jean, Mirza, Mehdi, Xu, Bing, Warde-Farley, David, Ozair, Sherjil, Courville, Aaron, and Bengio, Yoshua. Generative adversarial nets. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 27*, pp. 2672–2680. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

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

Pei, Jian, Han, Jiawei, Mortazavi-Asl, Behzad, Wang, Jianyong, Pinto, Helen, Chen, Qiming, and Hsu, Umeshwar DayalMei-Chun. Mining sequential patterns by pattern-growth: The prefixspan approach, Nov 2004. URL https://www.computer.org/csdl/trans/tk/2004/11/k1424.html.

Radford, Alec, Metz, Luke, and Chintala, Soumith. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015. URL http://arxiv.org/abs/1511.06434.

Weinberger, Kilian Q. and Saul, Lawrence K. Distance metric learning for large margin nearest neighbor classification. *J. Mach. Learn. Res.*, 10:207–244, June 2009. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1577069.1577078.