CS464 – Introduction to Machine Learning

Project Final Report

Handwriting Recognition Using Machine Learning Methods

Berdan Akyürek	21600904
Turan Mert Duran	21601418
Mannan Abdul	21801066
Maryam Shahid	21801344
Asım Güneş Üstünalp	21602271

Table of Contents

1. Introduction

Our lives are becoming increasingly intertwined with the digital world nowadays. And with this development, it is becoming essential that we are able to share information across the physical and digital aspects of our lives. Although handwriting is still in use, it is preferred to have digital versions of any text in terms of reliability, ease of storage, indexing and searching. For all these reasons, converting currently existing handwritten texts into digital text is an important problem to be solved. It is not possible to convert all handwritten text to digital manually. Also, handwriting recognition is a problem that has many different aspects and it is not possible to convert to an algorithm easily. Therefore, using a machine learning approach to solve this problem would be appropriate.

In this project the main aim was to test different machine learning methods learnt in this course to find a successful and efficient way to convert handwriting to digital text. In order to achieve this goal, a dataset having handwritten names and their labels is used. In order to try both word by word and letter by letter models, the images are cropped and segmented to letters. kNN, Naive Bayes, SVM and CRNN approaches are applied to the dataset.

2. Problem Description

Our goal in this project is to find ways of converting handwritten images to text and to do so accurately. Accuracy is a very important metric for us. This is because any model that is deployed to a commercial application needs to be extremely accurate, to be able to correctly recognise text from any images. If it does not do that, users may end up frustrated and not use the application. The first step to building the model is to analyse the dataset we have. It could give us a clue as to which direction we are supposed to lean in. Looking at the dataset, the first thing we immediately find out is that there is noise in the dataset, we have words in the image that are not in the label for that image. This tells us we need to clean the dataset. Another thing we need to decide is whether we go for methods that require us to segment the image into characters or try approaches that can recognise whole words.

The questions we try to answer in this project are: How good are the models at predictions and which one works best? Is our chosen model trained to generalize (can recognise unseen images fairly accurately)? What method of parsing the dataset works better? Using words or characters? What kind of surprises did the models' performance yield about relations in the dataset.

3. Methods

Dataset

Dataset that is used for this project is called “Handwriting Recognition, Transcriptions of 400,000 handwritten names”[1]. It contains around 400.000 handwritten words and their digital labels. It is around 1 GB. Also the data is already splitted into train validation and test sets. It consists of 80% train, 10% validation and test data. In the dataset website, its usability is marked as 9.4 out of 10. So although it is mostly usable, it still has some problems that make it hard to use. The main problem was the unwanted texts that appear frequently on the dataset. These unnecessary fields are considered as noise of the data that reduces the usability. Also the data contains poor quality images that are unreadable. These were the reasons that decreased the usability rate of the dataset. In order to overcome these problems, preprocessing was necessary.

Preprocessing

Before starting the training process, the dataset is examined. It is realized that data is not standard and some preprocessing may be necessary. An example data that needs preprocessing is as shown below.

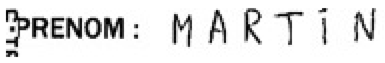

Figure 1.

The problem is, this photo is labeled as “MARTIN” but it contains the word “PRENOM” too. The data contains such photos in a great amount. So the data needed to be cropped to the necessary content. So, each image is cropped similar to the image below.

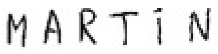

Figure 2.

In order to do this, Google Vision API is used[2]. This API simply detects each word in the image separately and gives coordinates of it in the image. We created .cropinfo files that contain information returned by Google API without doing actual cropping. We saved each word detected in each image and the coordinates of four corners of these words in images to these files to perform cropping later. Using these coordinates, it is possible to get rid of unnecessary words and non-text areas. This way, data is converted to more clean data and total data size is reduced.

By examining the cropinfos, it is realized that there are faulty results returned by Google API. Some text is undetected or wrong detected by Google API. For example, for the image below, no text is detected by Google.

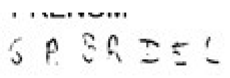

Figure 3.

For this situation, when no text is found, the image is discarded.

Also, for the image below, the text is detected as “BALTHA2AR” despite it is “BALTHAZAR”:

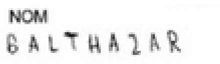

Figure 4.

For this situation, comparison is done in terms of approximate string matching. Instead of exact detection of a word, a similar word is also interpreted as correct.

Lastly, the following image is detected as three different words as “KE”, “V” and “IN” although it is a single word “KEVIN”:

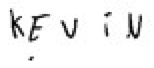

Figure 5.

For this situation, combinations of consecutive words are also checked.

Considering these edge cases and cropinfos, each image is cropped to the labeled content. Cropping is done on all train, validation and test sets. After cropping all the data, data size in terms of MBs is decreased to half due to cropping and discarding unsuccessful crops. The cropping quality could not be tested automatically, however by inspection, the results are evaluated successfully.

This cropped dataset is used with the initial model that does not need segmented letters. However, in order to test some letter by letter prediction models, segmented letter images were required. In order to implement models like kNN, further preprocessing is done. Each cropped image is cropped again to letters. For this, an algorithmic approach is used instead of using an API. Each image is first converted to pure black and white and gaps between letters are detected and cropping is done on the original image according to detected coordinates on pure black and white images. Since the dataset does not contain much cursive handwriting, this method is expected to work. Also, only vertical gaps are considered since there were not tilted or vertical written words in the dataset. After detecting and segmenting letters, each letter image is cropped from top and bottom again to eliminate gaps on top and bottom of letters. An example result of this process is shown below:

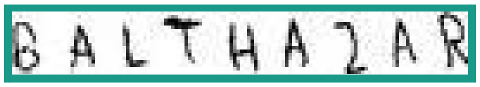

Figure 6.

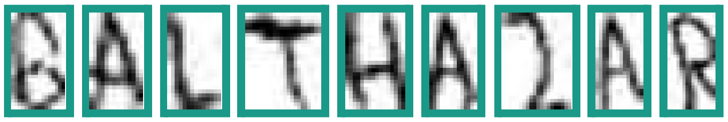

Figure 7.

After the cropping, the number of images is increased by around five times. However, only around 70% of the number of segmented letters were matching with the number of letters in the labels. In other words, for around 30% of the data, the number of letters in the labels were not matching with the number of letters detected by the letter segmentation algorithm. However, since it was not possible to use all of the data due to the big size of the data and the high train and test cost of models, it was not a limiting problem. The data that is used is selected from the ones that number of letters of labels matching with the number of letters detected from images. This way, it was intended to eliminate most of the faulty preprocessed data.

Lastly, data is standardized by converting all to the same size for both segmented and non segmented images and removing the data that is labeled as “Unreadable”.

CRNN Approach

Initially, preprocessing was kept to a minimum so we could focus on building a working model first. The pandas python library to read the CSV files containing the URLs of our images and their labels. The first step in preprocessing our images was to remove all such images whose labels were null. This step removed 565 images in the training set and 78 images in the validation set. Looking through the dataset, we also found some images that were labeled unreadable, so the next step was to remove these images from the dataset. These images were either cropped such that no words could be seen or were just illegible. We also set all labels to uppercase for 2 reasons. For uniformity in our labels but more importantly because all our images are of uppercase handwritten words.

After cleaning up the dataset, we read all the images into an array. For each image, we resize the image to width = 256 and height = 64 and then normalize the image values to be between 0 and 1. Resizing each image can also help us remove some of the unwanted text that is present in each image. Each image is read using the cv2.imread method from the cv2 library as a grayscale image. We read all the images in the training set using the procedure mentioned above into an array that has the training images and the validation images using the same procedure into an array that contains validation images. After the images are read, we reshape the image arrays so they can be fed into the CNN. We were not using the whole dataset but only 30,000 training images and 3,000 validation images. The reason was that when we reshaped the training array into a format we can input to the CNN, the training images array took on the shape of (num_images, 256, 64, 1). If num_images was the whole of the training dataset, the array needed a whopping 40.4 GiB to be stored in memory.

The model uses a Conv2D layer from tf.keras.layers as the first layer, with 32 filters and a kernel size of (3, 3). It then uses the BatchNormalization function to normalize our values as it is supposed to help the learning process run faster. We use a relu activation function with our convolving layer and MaxPooling2D function that has a (2, 2) pooling window which picks the highest value in that window and replaces the whole window with it. This helps us remove features in the image that are not useful and it will also remove the chances of the model overfitting as the model will not try to learn every single parameter. Two more convolving layers are added after with a dropout of 0.3 to remove the effects of overfitting further. After adding the convolving layers to the model, we add a layer to reshape the output from the previous layer and add a dense layer with 64 units and a relu activation function which means each of these 64 units are deeply connected (to every neuron of the preceding layer). So the dense layer can be thought of as consolidating the output of the CNN part of our model. We then use a Long

Short-Term Memory (LSTM) layer for the RNN part of our neural network. Research indicated that using a bidirectional LSTM layer can lead to better learning so that is what we are using because by doing this, our output layer can get information from both the past and future states. This is because the layer trains the model once on the input sequence as is and once on the input sequence reversed. RNNs are good for training models where sequences matter, so applications like speech recognition, etc. As mentioned before, we need this property because we have not segmented our images into characters for this approach so we cannot just classify based on that. After adding the RNN layers, we again use a dense layer with num_of_chars (in our alphabet) + 1 (for ctc blank) (total = 31) units with a softmax activation function. The output shape of our predictions is (64, 31) which means that the model will predict words of length 64 (to allow for enough ctc blanks) and will have the probability of each of the 31 alphabets that we defined earlier.

The backbone to this approach is the Connectionist Temporal Classification (CTC) Loss [4]. It helps remove the need to annotate the dataset on each timestep, which would be a deeply time-consuming task. But, CTC performs two more functions that are even more important. It helps train the model by providing a loss value for the model to reduce and it helps us decode the output of the CRNN.

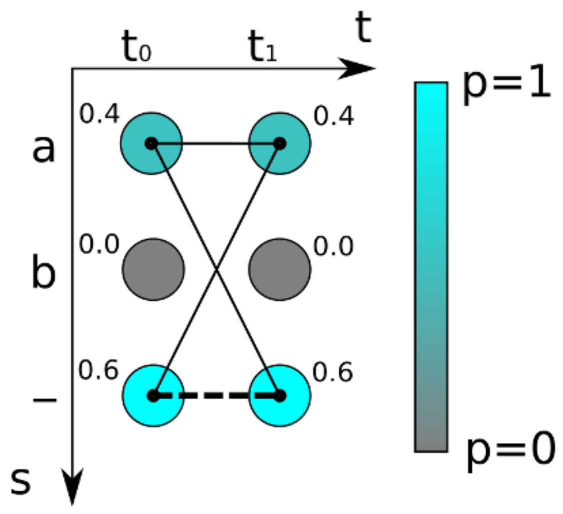

Figure 8: CTC loss calculation.

The above figure demonstrates how the ctc loss is calculated. The CTC function sums up the score of all possible alignments (paths) of the Ground Truth label; this means it does not matter where the text appears in the image. So for the example above, if our ground truth label is “a”, then we have 3 possible paths: “aa”, “a-”, and “-a”. In the CTC function if a character appears at 2 timesteps consecutively, like “aa”, it means that the letter a is stretched across both timesteps and not that there is an “a” on both timesteps. That scenario would be represented by a CTC blank “-”. When we get a sum of all possible path which would ideally be 1 meaning the text is definitely in the image, we can take the log of this sum and put a minus in front of it so that we can reframe this problem as a minimisation problem where we can train the CRNN to minimize this loss value.

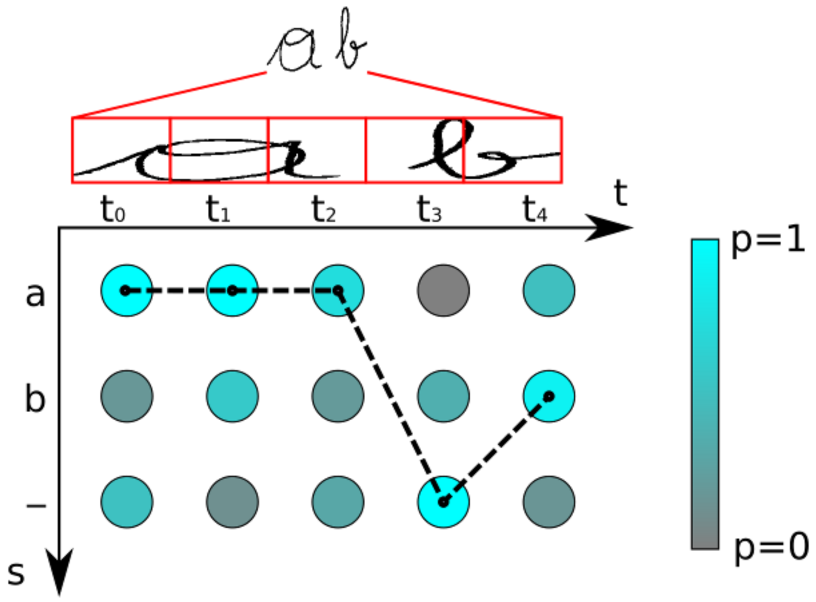

Figure 9: CTC Best Path Decoding.

On the other hand, CTC helps with decoding using a fairly simple method called Best Path Decoding. The character with the most probability at each timestep is chosen, at the end all the duplicate characters and CTC blanks are removed which leaves the recognised text at the end. An example is shown in the image above. While feedback on the progress report requested an equation for the CTC function, it is not possible because having read the paper, we would need to use at least a whole page to get the information required to understand the formulas across so a link is provided at the following reference [5].

This model also went through some improvements that need attention. As mentioned, we had access to a cleaned dataset. That is, it did not have words in the image that were not present in the labels. Some images were corrupted during the cleaning process so these images had to be dropped from the CSV file too. In this step, we also started using the whole dataset to be able to train the model. Tensorflow-gpu was also configured so the model could be trained on a gpu, this lead to a 10x improvement in training times which was a massive relief. To get around the constraint posed by the dataset size, we also used mini batch training with 2 different methods. Python generators were used in the first method and these generators were given to the model.fit() method to train the model. The generators would yield 128 training images and 16 validation images at a time. So in essence, our batch size was 128 and 16 for training and validation respectively. In the second method, we would gather 4096 training images in an array and 512 validation images in another array. The reason for these number of images is that these were the maximum number of images the gpu could train on without running out of memory. The model.fit() method was called in a for loop which basically means that the training would be done for 60 epochs on the first batch of 4096 images and then validated after each epoch using 512 validation images. After the training was done, we would load the next 4096 training and 512 validation images and train the model on these. We would continue like this until the dataset was exhausted. Both of these approaches were also complimented by an early stopping callback with a patience value of 10. This meant that if the validation loss did not go down for 10 epochs, the train would stop. We also had a model checkpoint callback that would save the weights for the models with the maximum validation accuracy and the lowest validation loss.

K-Nearest Neighbors

K-nearest neighbors classification is the second most accurate giver classifier among other classifiers when it is used for recognition of handwriting images [3]. It is easy to implement and gives good accuracy results. In order to apply k nearest neighbors to our dataset, images of handwritten texts should be firstly preprocessed by removing unwanted text in the images. After that, applying kNN would not be giving much accuracy when we train the classifier with images of words that are composed of lots of combined letters. Therefore, we needed one more preprocessing of images. Words should be divided into letters and for each letter there should be a new label table. After all these preprocessing processes had been done, we were ready to implement our kNN model. First of all, it was noticed that all segmented letter images are in different sizes. So, there needed one more preprocess which should be arranging all letters in the same dimensions. We decided that 32x32 for each segmented letter would be enough. Therefore all trained images and tested images are resized into 32x32 dimensions and their pixel datas are given into two dimensional arrays. Each 32x32 array is flattened into 1x1024 arrays and they are given into our kNN model. For our kNN model before the training data there should be a decision to choose which distance metric we use (Euclidean distance or Manhattan distance). By smaller datasets we tried both of them and none of them give much more accuracy than the other one. Therefore we have chosen the Euclidean distance metric to use. Our model is trained by 200.000 images of letters and tested by 15.000 images of letters. After segmenting letters from the words, we gathered much more letters like (2 million) but it is observed that increasing the number of images is not changing too much accuracy after some threshold. Therefore we decided to use 200.000 images. On the other hand, training with too much data requires too much time too. We could not train our kNN model by using all data that is preprocessed because of time constraints. Even trained kNN model with 1.000.000 images took much time and in kNN model when training data is increasing, time to predict each test image is increasing too. Therefore, even if we could train using all the data, we would not be able to test our images. Another constraint that should be taken into account was the n_neighbors value. It is the value that is used for predicting test values. We made some trials with different n values whose accuracy results can be seen below.

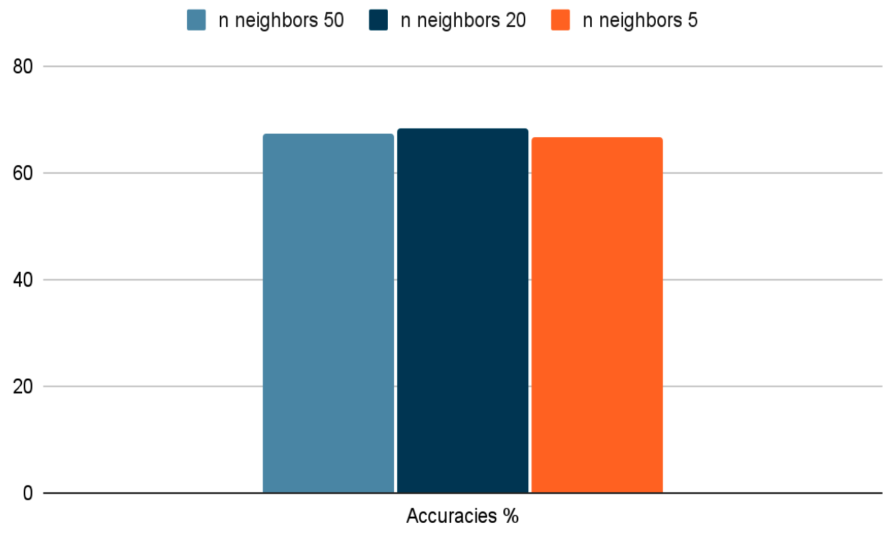

Figure 10: Accuracy Results of kNN with Different N_neighbors

After these trials are completed, it is observed that taking n_neighbors as 20 would be giving us most accurate results. Results of the tests will be discussed in the results section.

Supper Vector Machines (SVM)

SVM is known for its accurate results for pattern recognition using linear and nonlinear models. After researching similar datasets, we came across promising results of an SVM model for handwritten digits, so we decided to apply a similar model to our dataset [3]. Initially, images were preprocessed by cropping unwanted data in the images. Next, letter segmentation was carried out in order to use SVM. Further preprocessing was required since SVM requires input of an equal consistent size and the segmented letter varied in size from one another. We assumed that 32 x 32 would be appropriate. Thus, all images in the training, testing and validation dataset were resized into 32 x 32 images. The pixels were then converted into pixels and passed into a 2D numpy array. Then, an SVM classifier was initialized and trained using the training dataset. A low gamma value of 0.001 was used to limit biases. Since the dataset was unexceptionally large, the training dataset was sent into the classifier in batches. To get a fair rate of accuracy with respect to other models, in total the classifier was trained with 200,000 images of segmented letters and tested with 15,000 images of segmented letters out of the 2 billion images of letters we got after preprocessing and letter segmentation. The validation dataset of images was used to evaluate the model fit on the training dataset. The results of this model are discussed in the results section.

Gaussian Naive Bayes

Gaussian Naive Bayes is the least performative among the models we trained when recognizing handwritten digits[3]. We still wanted to try this model just to learn. As with the other models, the images were preprocessed before using training. Images contained unwanted words that were not present in the labels so we removed those. Then we segmented the letters to be able to train it with Gaussian Naive Bayes algorithm. Then the segmented letter images were all resized to 32x32 and flattened to ensure uniformity. Then the model has been trained using 200.000 segmented letter images and tested with 50.000 segmented letter images.

4. Results

CRNN Approach

Training methods	Character Accuracy	Word Accuracy
Initial	87.77%	73.63%
Noisy dataset + for loops	87.08%	76.15%
Clean dataset + for loops	80.40%	67.27%
Noisy dataset + generator	87.02%	77.87%
Clean dataset + generator	86.60%	77.49%

Table 1: accuracy values for all different training methods.

The constraints of the initial training method (noisy dataset, 10% of dataset used, cpu trained etc) have been discussed before. Noisy dataset is the dataset that has words that do not appear in the labels while the clean dataset does not include those words. The for loops and generator training methods have also been discussed in detail in the methods section.

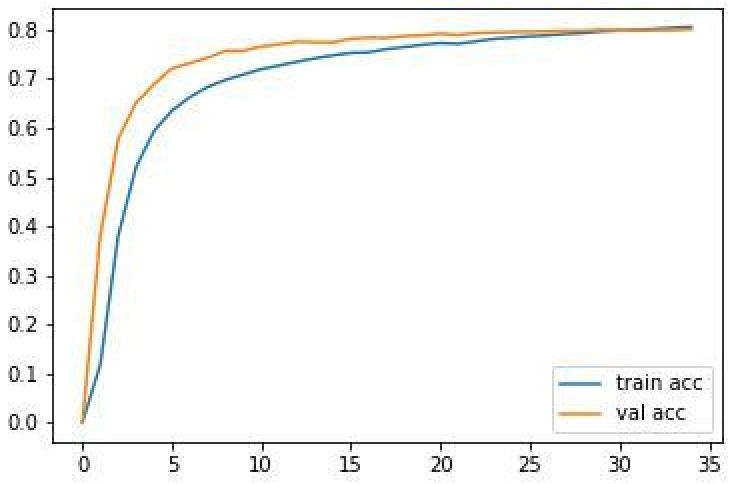

Figure 11: training accuracy vs validation accuracy.

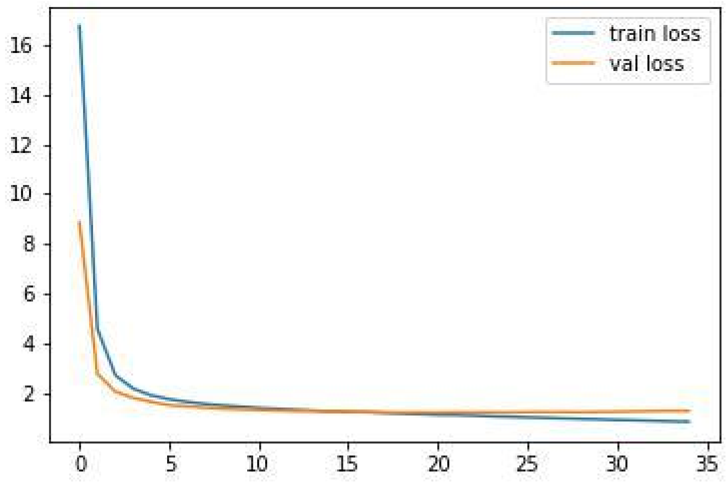

Figure 12: training loss vs validation loss.

Figure 18 and 19 are chosen for the best performing method, which is using the noisy dataset + python generators to go through the whole dataset in one epoch instead of training the model on mini-datasets in a sense. Seeing as I forgot to get the graphs for the other methods, it did not seem feasible to train with each method again to get the model history as each time training has taken around 8 - 9 hours making the total time for training a whopping 65 hours, when time is included for failed experiments and errors.

To answer the questions posed in the start of this report, the model seems to be quite good at making word predictions, specifically the noisy dataset + generators approach. It also seems to have generalized quite well as its test dataset performance was almost the same as its performance on the validation datasets. The character accuracy for the chosen model on the test set is 86.6% while the word accuracy is 77.49%. Testing was performed for this dataset because it had the highest validation accuracy out of all the other models that we tried. The model checkpoint callback was also useful because it preserved the best validation accuracy value of the model meaning that the patience value of early stopping did not need to be too small for fear that our model has started to overfit and it could be given the margin to get better if it was a temporary upwards trend in the validation loss.

K-Nearest Neighbors

kNN Model gave us %68.2 accuracy results when predicting letters when n_neighbors used 20 and 200.000 letter images used for to train and 40.000 test segmented letter images used for testing. When 100.000 segmented letter images used for train and same test data were used, accuracy was %56.4. As trained data increases, accuracy increases too. However it is observed that after some threshold it was not affected by trained image size. By using 300.000 segmented letters in the trained model, it gave us %69.3 accuracy results in 40.000 same tested data. It was not a tremendous difference as it is in 100.000-200.000. Therefore, we have chosen to use 200.000 letter images for training because using 300.000 trained images in knn model was taking too much to predict, etc. When we are predicting letters %68.2 is acceptable accuracy but when it comes to predicting a word, our model is most likely to mispredict one or more letters in a word. Therefore when we accept predicting BALTHAZAR as BALTHOZAR is a false prediction, our predicting word results are too low. Our model has more chances to predict truly in short words. Because in each letter, the chance of predicting the whole word truly is decreasing by %31.8. When we look at the results mathematically, the formula of predicting each word truly can be given like that. Let's say n is the number of letters in a word. Then our predicting truly chance by using our kNN model is : $\prod_{0}^{n}(0.682)$. We observed by our model that, predicting word accuracy as %12.3 that is expected.

Support Vector Machines (SVM)

The SVM model gave an accuracy of 65.46% while predicting letters. This model was trained with 300,000 images of segmented letters and tested with 40,000 images present in the testing dataset. This accuracy is given for letter prediction using SVM. However, while predicting words there were major drawbacks of the models and the accuracy dropped to a low 13.21%. The reason for this decrease is that while predicting letters - false prediction does not affect the accuracy by a large extent. However, while predicting words, even a single falsely predicted letter leads to an inaccurate prediction of the entire word. Thus, the accuracy rate of prediction words as a whole falls drastically.

Gaussian Naive Bayes

The Gaussian Naive Bayes model that we trained gave an accuracy score of %63.6 for predicting letters. The model was trained with 200.000 segmented letter images and tested with 50.000 segmented letter images. Based on the letter accuracy, a word accuracy has been calculated, it is ~%10. As explained above in other models, the reason for this low word accuracy is that in our average 5-letter words, even a single mispredicted letter makes the whole word mispredicted.

5. Discussion

Some things of note here about the CRNN approach. The for loop approach should have included a bias in the model towards the initial batches of the dataset as the weights were updated with them for 60 epochs. 1 thing that may have stopped this is the early stopping callback. In the latter parts of training, the model would only run for 13 to 16 epochs on each mini-dataset. With a patience value of 10, this means that only 3 to 6 epochs were useful in updating the model further. The generator approach did do slightly better though so that is why it was chosen as the best implementation for the CRNN approach and eventually as the best approach we came up with for this problem. The results as specified before are as follows: The character accuracy on the test set is 86.6% while the word accuracy is 77.49%. Using the whole dataset as opposed to 10% of it did not also yield significant improvements in the model performance with only a ~4% increase in the word accuracy for predictions. The most surprising thing though was that cleaning up the dataset actually reduced performance across both the for-loop and generators method and by significant margins too. This could suggest that the words not included in the labels but in the image may have some correlation to the words that appeared in the image and were not noise per se. This correlation also only pops up in the CRNN approach as we do not use whole images in any of our other methods.

A comparison of the other three models - Support Vector Machines (SVM), K-Nearest Neighbours, and Gaussian Naive Bayes is given below in figure 13. As noted, the highest accuracy is for K-Nearest Neighbours, followed by SVM and the lowest is for Gaussian Naive Bayes Model.

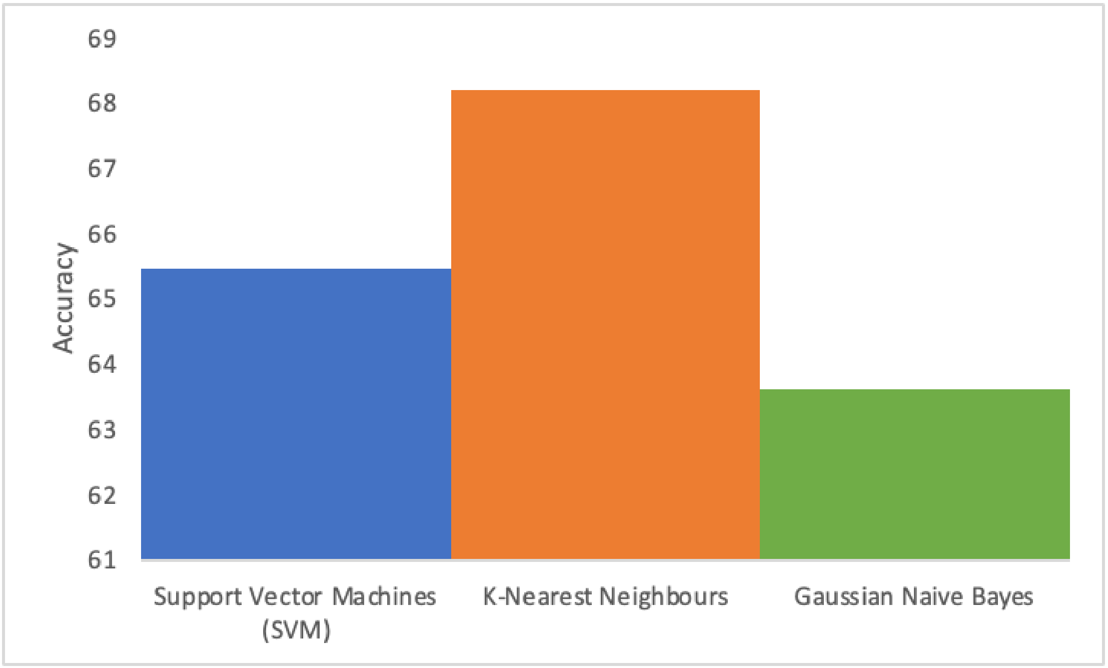

Figure 13: Comparison of Accuracies between SVM, KNN, and Gaussian Naive Bayes

A general problem with every model is also that our training dataset only includes images that have uppercase hand-written characters and so the models have been trained to recognise them accordingly, this means that it is unlikely that the models will perform very well if we ask it to recognise lowercase handwritten characters.

6. Conclusion

We learned a lot during our implementations of the models. One important thing is that things rarely go as you plan them and your expectations can be flipped on their heads very randomly (there being a correlation between the words in the labels and the words that were not in the labels).

One possible place that we would like to pursue is to try and find ways to see how the perceived noise in the dataset was actually helping the model perform better.

The data size became a limiting factor for some models. Although we had a big amount of data, it was not possible to use all of it. As the data size increases, time required to train/test models increases exponentially. This made it hard for us to use a big amount of data.

At the end of the project, it was concluded that the best model among all models tested were CRNN with 87% accuracy. However it is still far away from being ideal. More work must be done in order to come up with better ideas and find a more successful way of handwriting recognition.

7. Appendix

Work Contribution

Mannan Abdul: Designed and tested the CRNN model, worked on the presentation, the progress and final reports by explaining all parts concerning the CRNN model.

Turan Mert Duran: Designed and tested the kNN model, worked on the proposal, presentation, the progress and final reports and explained all parts of the kNN model.

Berdan Akyürek: Implemented preprocessing scripts related to cropping and letter segmentation. Managed the use of Google API and image segmentation algorithm. Splitted the data to group members for them to preprocess simultaneously. Contributed to the writing of the proposal, progress report and final report. Also contributed to the preparation of the final presentation. Presented preprocessing part of the final presentation.

Maryam Shahid: Designed and tested the SVM model. Conducted relevant preprocessing required for SVM. Contributed to the proposal report, progress report, presentation and final report by focusing on the SVM model and classifier.

Asım Güneş Üstünalp: Designed and tested the Gaussian Naive Bayes model. Worked on the proposal. Prepared the parts about Gaussian Naive Bayes model for the presentation, the progress and final reports.

References

[1]. "Handwriting Recognition", *Kaggle.com*, 2021. [Online]. Available: https://www.kaggle.com/landlord/handwriting-recognition. [Accessed: 25- Dec- 2021].

[2]. "Detect text in images | Cloud Vision API | Google Cloud", *Google Cloud*, 2021. [Online]. Available: https://cloud.google.com/vision/docs/ocr. [Accessed: 25- Dec- 2021].

[3]. "The Best Machine Learning Algorithm for Handwritten Digits Recognition", *Medium*, 2020. [Online]. Available: https://towardsdatascience.com/the-best-machine-learning-algorithm-for-handwritten-digits-recognition-2c6089ad8f09. [Accessed: 25- Dec- 2021].

[4]. "An Intuitive Explanation of Connectionist Temporal Classification", Available: https://towardsdatascience.com/intuitively-understanding-connectionist-temporal-classification-3797e43a86c [Accesed: 25-Dec-2021]

[5]. "Connectionist Temporal Classification: Labeling Unsegmented Sequence Data with Recurrent Neural Networks", Available: https://www.cs.toronto.edu/~graves/icml_2006.pdf [Accesed: 25-Dec-2021]