# COMP 551 Kaggle Competition Report

Team Name on Kaggle: MTL551

*Yanhao Gong*
*ID: 260546543*
*Email: yanhao.gong@mail.mcgill.ca*

*Kechen Qian*
*ID: 260559868*
*Email: kechen.qian@mail.mcgill.ca*

*Ehsan Nateghinia*
*ID: 260789004*
*Email: ehsan.nateghinia@mail.mcgill.ca*

***Abstract*—This report focuses on the Kaggle competition based on image analysis. The goal is to design a machine learning algorithm that can automatically identify hand drawn images. In this report, we start with the introduction of this project. Then we introduce the feature design and algorithms we used for comparison. At the end of this report, we compare all the results from the models and select the best one of them.**

## I. Introduction

For this project we are supposed to build a model to classify hand drawn images. The dataset is a subset of Googles quick draw dataset. It has 31 labels and one of them is empty class. Then a significant noise (not uniform) has been added to dataset, which makes it more complicated. The pictures are also randomly placed in a $100 \times 100$ grid. Preprocessing of data is needed to eliminate the noise in the dataset and it could also increase the accuracy for feature selections. The details could be seen in section II. We choose several algorithms, LinearSVM, Feed-forward Neural Network, KNN(k-nearest neighbors), XGboost and CNN(Convolutional Neural Network). We take LinearSVM as our baseline learner. All brief introductions of above algorithms could be found in section III. In section IV, we introduce the methodology of all the algorithms, including the decisions about training/validation split, regularization strategy if necessary, and the choices of tuning all the hyper-parameters.

## II. Feature Design

### A. Pre-Processing

Figure 1 shows nine samples taken from the training image set. It can be observed that the location of each hand drawn are placed in the $100 \times 100$ grid randomly. Additionally, non-uniform noises have been added to the pictures, where in some cases it is hard to find the real objects or at-least its parts.
Since the noises are not uniform, if we use image filters like Gaussian, it will also degrade the real object in the image. By looking at the images, we find out objects are mostly single connected hand drawns. Therefore, a simple segmentation algorithm could help us to extract the object. Since the background is zero, we use the method called skimage.measure.label, which gives the same label to all the connected pixels. In this case, we can extract the label with the largest frequency.

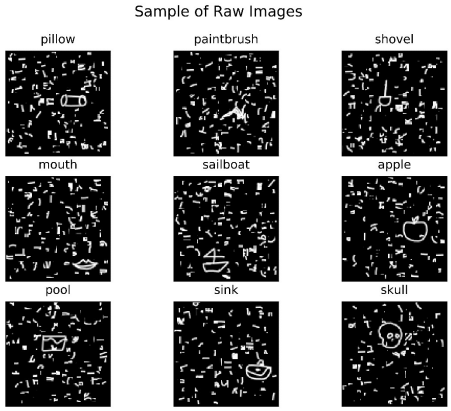


Fig. 1: Sample of Raw images

Figure 2 and Figure 3 show two examples of extracting objects from raw image. The first picture at the top-left is the original image. This image is binarized with a threshold value 5. The pixels with the intensities less than the threshold are set to zero and those with the intensities more than the threshold are set to one. The second picture of the first row shows the binarized image. Then, the connected components labeling algorithm is applied to the binarized image and gives the segmented image. As we can see every object has a single individual intensity.

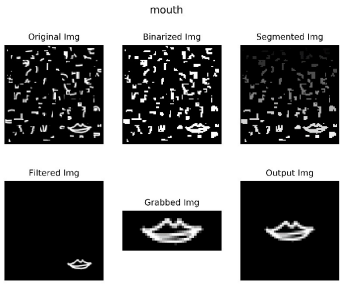


Fig. 2: Pre-processing Method

The second row of the Figure 2 shows the filtered object. Then this image is extracted with a bit bigger window (the middle image). After that, we have applied connected com-

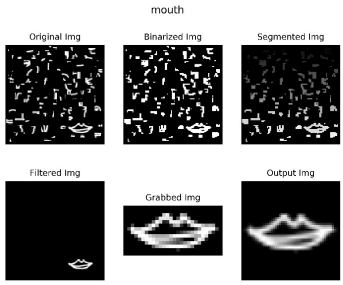


Fig. 3: Pre-processing Method

ponent labeling and have extracted any object with the size of greater than 40 pixels. In fact, an object might consist of several unconnected parts, and we have assumed that the smallest part is greater than 40 pixels. Tuning this threshold is very complicated, since we have 10,000 training images and we can not verify its performance. Therefore, we accepted its error at this stage, and will build our classification algorithms. At the last step, we have placed the obtained object at the center of a $50 \times 50$ image. Therefore, all the output images are $50 \times 50$.
Figure 3 has the same steps, except the last step where the image is re-sized to the $50 \times 50$ grid with the same vertical and horizontal ratio. Now that we get the image and have placed in the center of the output image, we can use it for classification algorithms.

### B. Feature Extraction

After obtaining each hand drawn, we decide to remove noises firstly and then extract SIFT, SURF and HOG features, but none of them has noticeable contribution on the classification performance. Then we flattened each image into a $1 \times 2500$ instead of $50 \times 50$. Therefore, we have 10000 training examples with 2500 features for each, and 10000 test examples as well. It should be noted that all the intensity values are normalized by dividing by the maximum intensity value. Additionally, normalized intensity values have been quantized or binarized for further processing. In the quantization method, intensity values (which are in range of 0 to 1) are quantized to 20 distinct levels. For example, 0.001 is quantized to 0 and 0.0513 is quantized to 0.05. Therefore, we have 21 distinct values in feature set: $\{0, 0.05, 0.1, 0.15, \cdots, 0.95, 1\}$. The number of quantization level is also a hyper-parameter, but after several simulations, we find that 20 works fine. In the other method, instead of quantization, the intensity values are binarized with the threshold of 0.1. Therefore, if the intensity is below 0.1 it will be set to 0 and if the intensity is greater than that, it will be set to 1. In fact, in this method, we give the same significance to most of the pixels in the hand drawn. To recap, we have four set of pre-processing:

- Preprocessing Method 1 (without scaling) followed by the Quantization
- Preprocessing Method 1 (without scaling) followed by the Binarization
- Preprocessing Method 2 (with scaling) followed by the Quantization
- Preprocessing Method 2 (with scaling) followed by the Binarization

These four conditions along with the hyper-parameters of each classification model require a lot of simulations and tuning. Firstly, we try some of the baseline with all four categories of preprocessing and then we continue with the best one for the rest of the project.

## III. Algorithms

We have applied different classification algorithms including LinearSVM, KNN, Neural Network, XGBoost and CNN. In this section, we will discuss them one by one.

### A. Baseline Linear Learner

*1) LinearSVM:* Support vector machines are supervised learning models with associated learning algorithms that analyze data used for classification analysis. LinearSVM is one of SVM algorithms to solve multi-class classification problems [1]. In our case, we used the build-in method called sklearn.svm.LinearSVC in python. Since this classifier is very basic, we take LinearSVM as our baseline method. The process of tuning the hyper-parameters of LinearSVM could be found in section IV.

### B. Feed-forward Neural Network

The Feed-forward Neural Network is a traditional neural network which has multiple fully-connected layers of neurons between input and output [2]. In this network, we have one hidden and one output layers and they are fully connected in our case. The weights between layers are trained with backward propagation. In this project, we implement the algorithm discuessed in the class (using sigmoid activation function) by ourselves, and the details could be found in section IV.

### C. Other Choices of ML Methods

*1) KNN:* K-nearest neighbors algorithm (KNN) is a non-parametric method used for classification [3]. The only parameter in KNN is the number neighbors(K). We tune different values of K in section IV.

*2) XGBoost:* XGBoost is an implementation of gradient boosted decision trees designed for speed and performance [4]. Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final predication by using a gradient descent algorithm. The XGBoost module is able to load data from various format generated by modules like NumPy and SciPy. In our case, train and validation data are in the form of numpy 2D array after applying the pre-processing method. XGBboost will accept those datasets and store them into Data Matrix together with the class of each input sample. After that, multiple hyper-parameters, for instance, max-depth

and learning rate, will be defined before starting to train the input data-set. The detailed process of splitting train/validation data sets and choosing hyper-parameters will be explained in section IV and find the optimal solution for K.

*3) Convolutional Neural Network:* Convolutional Neural Network (CNN) is a class of deep, feed-forward artificial neural networks, most commonly applied to analyzing visual imagery [5]. CNN generally is consist of convolutional, pooling, normalizing, fully connected (FC) and accelerating layers. Our CNN is built by using the Keras library, which runs on top of the tensorflow, and contains 4 2D convolutional layers, 5 normalization layers, 2 pooling layers and 4 FC layers. This model is evaluated by accuracy metric together with categorical_crossentropy losses since we have more than 2 output classes. Details including optimization and hyper-parameters tuning will be introduced in section IV.

## IV. Methodology

When splitting training and validation sets, two concerns should be paid attention to:

- With less training data, our parameter estimates will have relatively greater variance,
- With less validation data, our performance statistic will have relatively greater bias.

Broadly speaking, taking these two concerns into consideration, we should do more with the absolute number of samples in each category other than the percentage. However, considering that we have 10000 image samples in our training set, the choice of either 80:20 split or 90:10 split will not necessarily affect the variance in our estimation of hyper-parameters. Wherefore, based on Pareto principle, we start with the 80:20 split and subsequently change to 90:10 split since the latter has a relatively higher accuracy on validation data set with same hyper-parameters set.
The following sub-sections introduce our regularization strategy, optimization and hyper-parameters tuning for different algorithms.

### A. Baseline Linear Learner

*1) LinearSVM:* We apply four pre-processing methods which are explained in section II to generate four different datasets. The details could be found in section II. The hyper-parameter we choose for LinearSVM is C, penalty parameter of the error term. We tune different values of C including $\{0.001, 0.01, 0.1, 0.5, 1\}$ to exploit the best accuracy.

### B. Feedforward Neural Network

In this project, we implement the construction of the neural network by ourselves. We use one the hidden layer with 50 neurons. The output layer consists of 31 neurons since we have 31 classes. In this case we have two weights which are W0 and W1 respectively. The size of W0 is $2500 \times 50$ and the size for W1 is $50 \times 31$.

### C. Other Choices of ML Methods

*1) KNN:* Tuning is the first step, we evaluate the KNN with four different sets of data. We apply four pre-processing methods which are explained in section II to generate different datasets, including the dataset with or without scaling and quantization and the dataset with or without scaling and binarization. Next step, we tune the hyper-parameter of this classifier which is K (number of neighbors: $\{1, 2, 5, 10, 15, 20, 100\}$) to find out the optimal solution K among all these datasets.The results are shown in section V.

*2) XGBoost:* The hyper-parameters we define before model training process are listed in Table I.

TABLE I: Hyper-parameters for XGBoost

| Hyper-parameters | Value |
|---|---|
| objective | multi:softmax |
| eta(Learning rate) | [0.1, 0.2, 0.3, 0.4, 0.5] |
| max_depth | [4, 6, 8, 10, 12] |
| gamma | [0.1, 0.2, 0.3, 0.4, 0.5] |
| $n_thread$ | 4 |
| num_classes | 31 |

Among those hyper-parameters, we choose 'eta' and 'max_depth' to be tuned since others either have no effect on performance or decided by system specs. The default values of 'eta' and 'max_depth' are 6 and 0.3 respectively. Based on that, the tuning range of them are linearly expanded in both increasing and decreasing direction. We also add regularization to this method which is the hyper-parameter 'gamma'. The value of gamma is picked by tuning itself with the best performance achieved after tuning eta and max_depth with gamma equals to 0.

*3) Convolutional Neural Network:* Since we have 4 different sets of preprocessing method can be applied to generate input data for our CNN model, the decision is made by running each of them paired with CNN with exactly the same hyper-parameters setting. The performance is boosted by using a combination of preprocessing method 1 and binarization.
To optimize the performance of CNN, we can dig in two aspects: input data sets and hyper-parameters tuning.
For trick on input data set, since we are not allowed to use any supplementary data to enrich the training set, we introduce the image augmentation from keras library into out algorithm, named as "ImageGenerator". Basically, this built in function is able to artificially create training images through different ways of processing or combination of multiple processing, such as random rotation, shifts, shear and flips, etc to boost the performance of CNN. The existence of image augmentation significantly improve the accuracy, which will be discussed in section V.
In terms of hyper-parameters tuning, we divide the task into two sub-groups, which are hyper-parameters of CNN model and Image augmentation. The hyper-parameters chosen to be tuned are: number of filters for each layer, regularization rate, epoch, batch size, rotation angle and shift range of images. Since the computation complexity of simultaneously tuning

all parameters is unacceptable based on resources we have, a fixed number of each parameters are assumed in the first place and then start to tune them one by one. The starting values of hyper-parameters are reported in Table II. The number of neurons on last FC (Fully connected) layer should be equal to number of classes, which is 31.

TABLE II: Hyper-parameters for CNN

| Hyper-parameters | Value |
|---|---|
| No of filters in first two Con | 30 |
| No of filters in second two Con | 15 |
| kernel size | (3, 3) |
| pooling size | (2, 2) |
| No of neruons in 1st FC | 128 |
| No of neruons in 2nd FC | 50 |
| Dropout of Con | 0.75 |
| Dropout of FC | 0.25 |

## V. Results

### A. Baseline Linear Learner

*1) LinearSVM:* We tune the hyper-parameter C based on $\{0.001, 0.01, 0.1, 0.5, 1\}$ and all the results are shown in Figure 4 and Figure 5. We have used a 5-fold cross-validation, therefore each value is the average of the five different experiments. Figure 4 represents the results from the first preprocessing method, the plot at the left side is the result from quantization and the plot at the right is the result from binarization. In the same way, Figure 5 represents the results from the second preprocessing method, the plot at the left side is the result from quantization and the plot at the right is the result from binarization. From table III, we could see that The best results were achieved when using second preprocessing (with scaling and binarization) with C equal to 0.001. The average F1-Score for the validation set is about 0.361 and the average accuracy is about 0.389 (5-fold cross-validation).

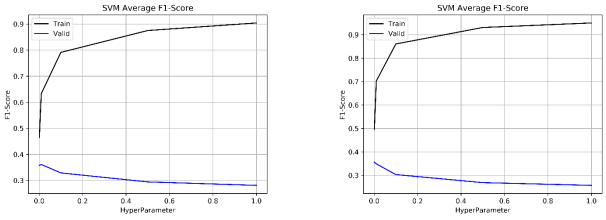


Fig. 4: SVM Average F1 score Preprocess 1

### B. Feed-forward Neural Network

The Table IV shows the performance of the implemented Neural Network containing one hidden layer with 50 neurons and one output layer with 31 neurons. The learning rate was tuned to 0.001 and these results were obtained after 5000 epochs. As we can see the Neural Network performance is very close to the LinearSVM. Since we are not assuming the pixel relations and giving flattened image to these algorithms, their performance is not very exciting and explains that why flattened images are not a good choice.

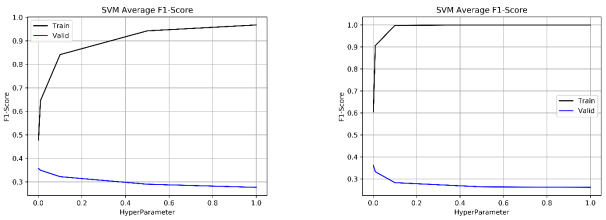


Fig. 5: SVM Average F1 score Preprocess 2

TABLE III: The results of the tuned LinearSVM for four different sets of data

| Preprocessing Normalization | Train F1-Score | Valid F1-Score | Hyper-parameter: C |
|---|---|---|---|
| 1-Quantization | 0.633 | 0.361 | 0.01 |
| 1-Binarization | 0.496 | 0.356 | 0.001 |
| 2-Quantization | 0.477 | 0.356 | 0.001 |
| 2-Binarization | 0.605 | 0.361 | 0.001 |
| Preprocessing Normalization | Train Accuracy | Valid Accuracy | Hyper-parameter: C |
| 1-Quantization | 0.642 | 0.386 | 0.01 |
| 1-Binarization | 0.522 | 0.391 | 0.001 |
| 2-Quantization | 0.499 | 0.387 | 0.001 |
| 2-Binarization | 0.617 | 0.389 | 0.001 |

TABLE IV: Performance of Neural Network

| Preprocessing Normalization | Train F1Score | Valid F1Score | Train Accuracy | Valid Accuray |
|---|---|---|---|---|
| 2-Quantization | 0.61 | 0.38 | 0.57 | 0.35 |

### C. Other Choices of ML Methods

*1) KNN:* Figure 6 and Figure 7 show the F1-Score of the hyper-parameters tuning with four different preprocessed datasets. Figure 6 represents the results from the first preprocessing method, the plot at the left side is the result from quantization and the plot at the right is the result from binarization. Also, Figure 7 represents the results from the second preprocessing methodthe plot at the left side is the result from quantization and the plot at the right is the result from binarization. We also use 5-fold cross-validation as we mention before.As we can see, by increasing K, the F1-scores drops for both training and validation sets. While, it seems that in K=1, we have overfitting, and the best K is between 5 and 10. The following table V shows the results of the tuned algorithm. The best results were achieved when using second preprocessing (with scaling) and binarization with K equal to 10. The average F1-Score for the validation set is about 0.459 and the average accuracy is about 0.475 (5-fold cross-validation).

*2) XGBoost:* The pre-processing set used for XGBoost is method 1(without scaling) and binarization. The best performance of XGBoost can be obtained after tuning all three

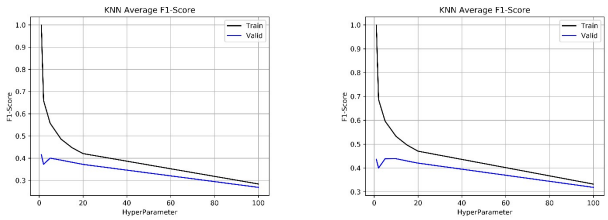


Fig. 6: KNN Average F1 score Processing 1

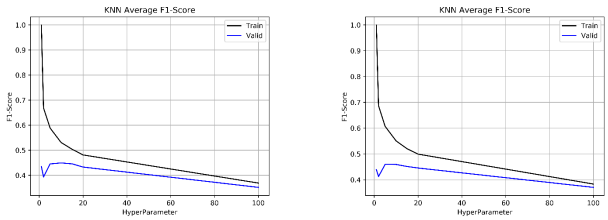


Fig. 7: KNN Average F1 score Processing 2

hyper-parameters mentioned in section IV-C2. The first tuned hyper-parameter is learning rate, known as 'eta'. Figure 8 shows the trend when tuning and the best result is achieved by $eta = 0.1$.

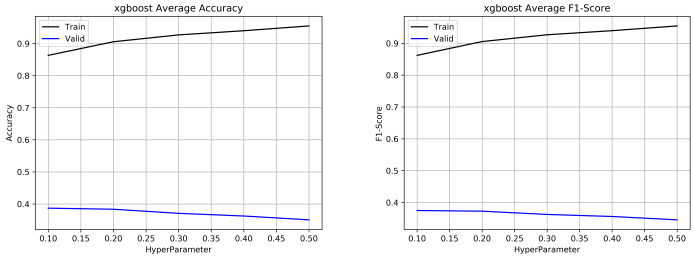


Fig. 8: XGBoost Average Accuracy

The best value of max_depth is 8 based on result trend shown in Figure 9. According to Figure 10, gamma is set to be 0.1 since the variation of gamma has no huge impact on performance. The performance is reported in Table VI.

TABLE V: The results of the tuned KNN for four different set of data

| Preprocessing Normalization | Train F1-Score | Valid F1-Score | K |
|---|---|---|---|
| 1-Quantization | 0.558 | 0.4 | 5 |
| 1-Binarization | 0.533 | 0.439 | 10 |
| 2-Quantization | 0.531 | 0.449 | 10 |
| 2-Binarization | 0.551 | 0.459 | 10 |
| Preprocessing Normalization | Train Accuracy | Valid Accuracy | K |
| 1-Quantization | 0.568 | 0.422 | 5 |
| 1-Binarization | 0.548 | 0.462 | 10 |
| 2-Quantization | 0.542 | 0.466 | 10 |
| 2-Binarization | 0.562 | 0.475 | 10 |

TABLE VI: Performance of XGBoost

| Preprocessing Normalization | Train F1Score | Valid F1Score | Train Accuracy | Valid Accuray |
|---|---|---|---|---|
| 2-Quantization | 0.86 | 0.374 | 0.861 | 0.387 |

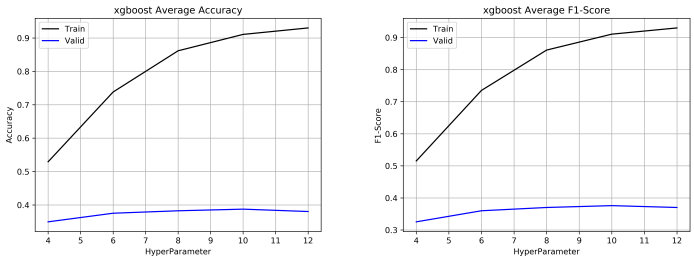


Fig. 9: XGBoost Average Accuracy

*3) Convolutional Neural Network:* As we mentioned in section IV, the best performance can be achieved by using the combination of preprocessing method 1 and binarization. Before adding the image augmentation, we tune the hyper-parameters such as number of filters in each convolutinal and neurons in first two FC layers, as well as dropout and batch size and the overall performance is stuck at 0.74. Even multiple times of tuning is made to, no huge improvement in performance is achieved. Therefore, we add the functionality of image augmentation. with multiple times of implementing, the values of ImageGenerator hyper-parameters are found to have no huge influence on performance if we keep them in a certain range. Table VII reports the final parameters of ImageGenertor we used in our algorithm.

TABLE VII: Final parameters of ImageGenerator

| Hyper-parameters | Value |
|---|---|
| rotation_range | 15 |
| width_shift_range | 0.1 |
| shear_range | 0.3 |
| height_shift_range | 0.1 |
| zoom_range | 0.1 |

Follow on, we start to re-tune the hyper-parameters of CNN model with the existence of image augmentation. Due to the limitation of report space, all hyper-parameters tuning can not

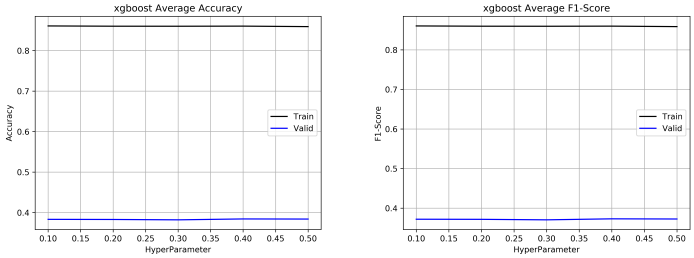


Fig. 10: XGBoost Average Accuracy

be listed here, Figure 11 shows a example of the trend of performance while tuning the number of filter size in FC layers and dropout rate.

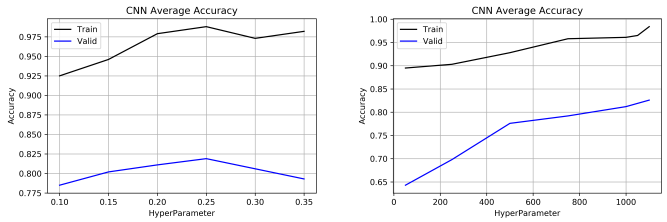


Fig. 11: CNN Accuracy

We apply the same strategy to all the other parameters and the final value used are reported in Table VIII. Since we use the callback function in our CNN model, whenever the accuracy on validation has a increment less than 0.001, the traning will automatically stop. Overall, the highest performance on Kaggle website we achieved is 0.83200.

TABLE VIII: Final Hyper-parameters for CNN

| Hyper-parameters | Value |
|---|---|
| No of filters in first two Con | 32 |
| No of filters in second two Con | 64 |
| kernel size | (3, 3) |
| pooling size | (2, 2) |
| No of neruons in 1st FC | 1024 |
| No of neruons in 2nd FC | 1024 |
| Dropout of Con | 0.25 |
| Dropout of FC | 0.5 |
| Batch size | 55 |
| epoch(when lr=0.001) | 40 |
| epoch(when lr=0.0001) | 500 |

### *D. Comparison between different algorithms*

The tuninng of hyper-parameters is shown in all the tables we mentioned above in this section. Among the accuracy of all the models we offer, we could find that CNN has the best performance which is 0.83200. For other algorithms, the accuracy is not that good. The accuracy of neural networks is around 0.35.Our baseline, which is LinearSVM, only has an accuracy of 0.389. All of them are below 0.5. The reasons are also explained in the result of each model above.

## VI. Discussion

We use several classifiers to do the image classification in this project. LinearSVM is a very simple classifier. However, with the increment of the dataset, its accuracy would drop rapidly. This is not a good choice for image classification. Feedforward network is a good classifier but in our case, we only use 2 layers and the feature size is very large. Without the convolutional layer, the accuracy could not be guaranteed. Above all, CNN is the best choice for this project and with the help of build-in method, we achieve the decent accuracy at the end. The main reason is that the convolution layers of the CNN apply multiple image filters (kernels) to find out the relation between pixels in different directions. It is equivalent to extract features from the images by using typical filters such as edge detector or Gaussian filter, and then give the extracted feature vector to the neural network. In fact, CNN will do these two stages together, and by adding multiple convolution layers, we can extract any information in the images. During this project, we compare all the models provided above and choose CNN as the best classifier. Our feature plan is to keep focusing on the construction of CNN as well as its hyper-parameters. From our exploration, we find the accuracy would increase when we increase the number of convolutional layers and choose the optimal number of neurons in the fully connected layers. However, the design exploration process is limited by the resource we have. Hopefully, a higher accuracy can be achieved by playing with those hyper-parameters and the CNN model itself.

## Statement of Contributions

Every one in the group contributed to this project. We defined the whole problem together and set up several meetings to talk about the details of all the procedures. Ehsan did a great job in feature preprocessing, and tried mutiple of ways to gather better features. For the model parts, Yanhao and Kechen designed the construction of CNN, as well as choosing the right hyper-parameters. Yanhao tuned the hyper-parameters to achieve the best accuracy on AWS. Ehsan was also working on CNN and implemented the algorithm for the feedfoward neural network. Additionally, Yanhao also provided another choice of model called XGBoost. We did the simulation together to gather all the results in this report, including the build-in methods(SVM, KNN). For the report part, we worked on our own part and helped each other to combine all the contents. Yanhao did the CNN and XGboost part. Kechen was responsible for abstract, introduction, SVM, NN and KNN parts. Ehsan worte the contents for data preprocessing, SVM, KNN and feedfoward NN. We did the results and discussion parts together to make the comparisons between all the models sufficient. We hereby state that all the work presented in this report is that of the authors.

## References

[1] ”Support vector machine”, En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Support_vector_machine. [Accessed: 26- Nov- 2018]

[2] [3]”Feedforward neural network”, En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Feedforward_neural_network. [Accessed: 26- Nov- 2018]

[3] ”K-nearest neighbors algorithm”, En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm. [Accessed: 26- Nov - 2018]

[4] Brownlee, J. (2018). A Gentle Introduction to XGBoost for Applied Machine Learning. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/ [Accessed 26 Nov. 2018].

[5] En.wikipedia.org. (2018). Convolutional neural network. [online] Available at: https://en.wikipedia.org/wiki/Convolutional_neural_network [Accessed 26 Nov. 2018].