1. 1.1 Classification: Feature Extraction + Classical Methods

   - Explanation of Design and Implementation Choices of your model:
     - For the first part of the project, we chose several supervised training methods, which include KNN, SVM, Random Forest, and Gradient Boosting, that we learned and already familiar with as the fundamental standpoint. For an extension, we choose to use an advanced training method called XGBoost. During the experiment, we discovered that the training process for some of the classifiers, like the Gradient Boost, is tedious and may never finish due to certain limitations on equipment. Thus, we choose to use XGBoost seeking to improve the ultimate accuracy on prediction, but also time efficiency on the overall performance on training, fitting, and predicting. A detailed description is available to further down in this report. So far in this assignment, two of the most suitable classifiers are SVM and XGBClassifier.
       Below is a one-sentence brief explanation of how each method works.
       * KNN: classify data sets with clusters of data that are close to each other with a specified range.
       * SVM: SVM finds a hyperplane with a soft or hard margin that best splits features into different domains. Radial basis function kernel (RBF), a nonlinear kernel, is used for this data set for it is not at all linearly separable. Figure 1 and 2 demonstrates a visual representation of the data set and reinforce the necessity of using RBF.

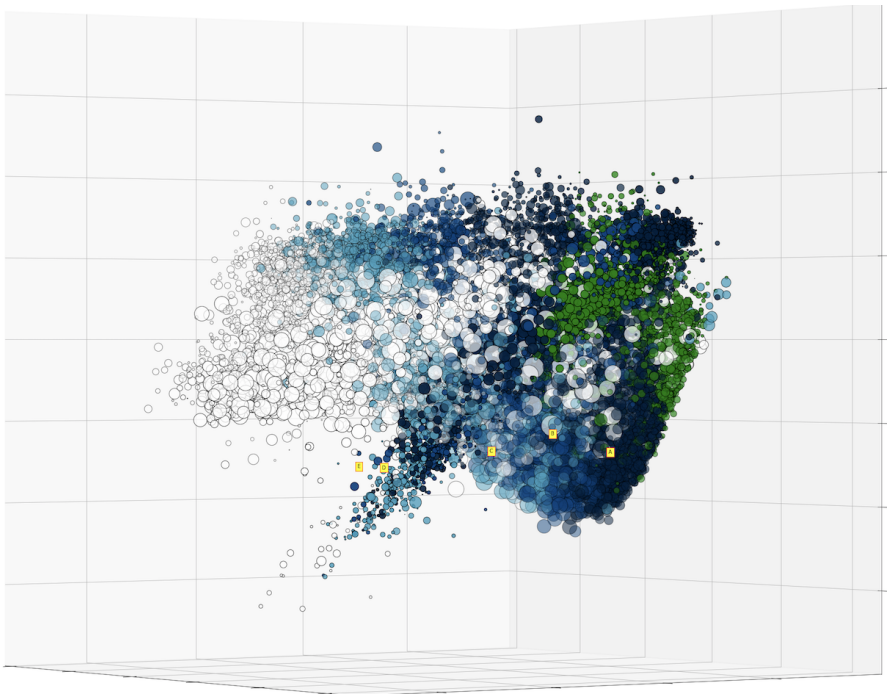


Figure 1: Cluster of Original data

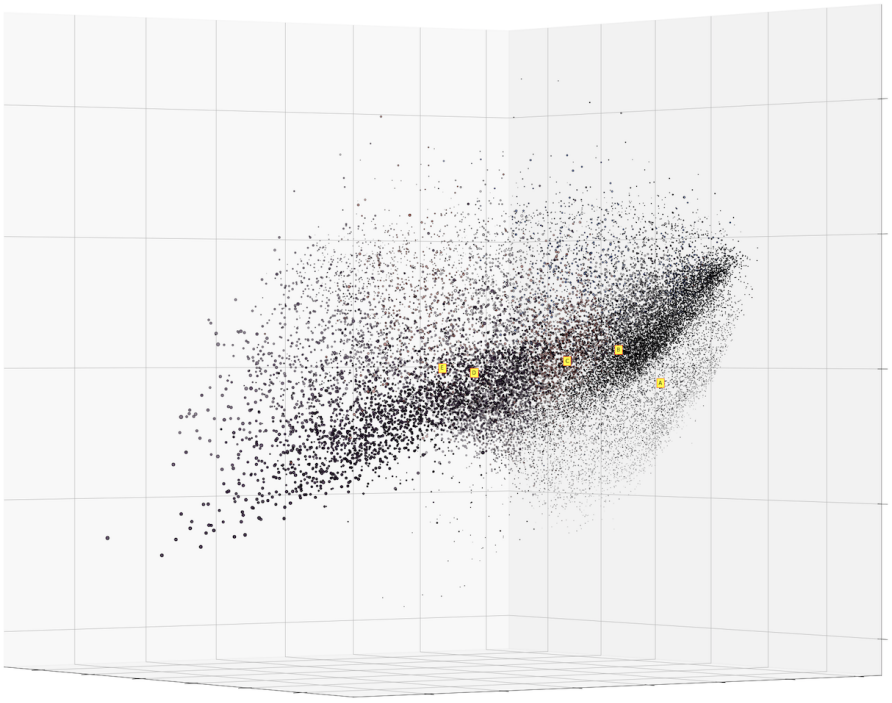


Figure 2: Cluster of standardised data

    * Random Forest: In prediction, random forest counts the vote from all decision trees and takes the majority. When there is no limit on the max depths, the tree would eventually fit all data provided and ensure the accuracy to be 100%. This must be avoided to prevent overfitting in prediction.
    * Gradient Boosting: Gradient Tree Boosting is stronger in prediction noisy data than random forester because it uses the loss from the previous iteration as the start point for the next iteration. This feature is helpful in a dense data cluster like this one, but the process is prolonged that only allows us to reach a quarter of the entire training process.
    * XGBoost: Assume Gradient Boosting method has the highest potential to identify the best prediction, but we lack the time and equipment for it to finish the train. XGBoost is used as a replacement, but also an update from it. XGBoost is an optimized distributed gradient boosting library designed to be highly efficient which our research can benefit from.

– For feature extraction, we applied PCA and LDA methods on each classifier for comparison. PCA projects a data set to a new dimension and reduces dimension by extract components with the highest variance. Although the principle of LDA is similar to PCA, it is a supervised feature extraction method. After projection, LDA prefers vectors with a low variance with class and high variance between classes. As you know, this data set contains a total of five different classes. Figures below are a partial demonstration of

each class.

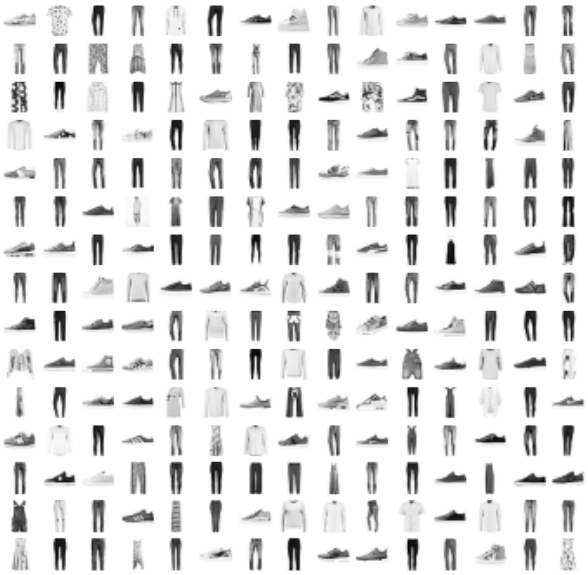


Figure 3: Class 0

Figure 4: Class 1

Figure 5: Class 2

Figure 6: Class 3

Figure 7: Class 4

Based on observation, all classes each a set of contains multiple clothing types, and all sets are identical between groups. As an assumption, the initial classifier uses color, greyscale color, to classify the data set. If the assumption stands still, a supervised feature extraction method like LDA will not be optimal for there is a relevantly low variance between classes. Accordingly, PCA is the chosen feature extraction method in the following report. The number of principal components may vary to finalize a suitable model for prediction.

- 1.2 Implementation of Design Choice

  In this section, we will briefly introduce the process of the overall experiment and show partial corresponding code blocks. The data is first standardized and projected using PCA and LDA. Different classifiers are built with the newly extracted dataset. The result comparison on the performance of each model, including run time and accuracy, will be further introduced below.

  – Packages used.

```
import numpy as np
import pandas as pd
import random
import seaborn as sns
import time
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

  – Feature extraction methods

```
#read data
scaler = StandardScaler()
train_set= pd.read_csv("train.csv",sep = ',',index_col = 0)
test_set= pd.read_csv("testX.csv",sep = ',',index_col = 0)
print(test_set)
#copy a new train set to analysis
temp_train = train_set[:]
temp_test = test_set[:]
del temp_train['Label']
X = temp_train.values
y = temp_test.values
#use normalization
X_std = scaler.fit_transform(X)
y_std = scaler.fit_transform(y)
#fit pca
pca = PCA(n_components=100,random_state = 42)
start_train_pca = time.time_ns()
train = pca.fit_transform(X_std)
end_train_pca = time.time_ns()
time_train_pca = (end_train_pca-start_train_pca)/1000000000
print("PCA train time:{}s".format(time_train_pca))
start_test_pca = time.time_ns()
test = pca.transform(y_std)
end_test_pca = time.time_ns()
time_test_pca = (end_test_pca-start_test_pca)/1000000000
print("PCA test time:{}s".format(time_test_pca))
```

Figure 8: PCA

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
scaler = StandardScaler()
#read data
train_set_lda= pd.read_csv("train.csv",sep = ',',index_col = 0)
test_set_lda= pd.read_csv("testX.csv",sep = ',',index_col = 0)
temp_train_lda = train_set_lda[:]
temp_test_lda = test_set_lda[:]
del temp_train_lda['Label']
X_lda = temp_train_lda.values
y_lda = temp_test_lda.values

#standardize and fit lda
X_std_lda = scaler.fit_transform(X_lda)
y_std_lda = scaler.fit_transform(y_lda)
lda = LinearDiscriminantAnalysis()
start_train_lda = time.time_ns()
train_lda = lda.fit_transform(X_std_lda,train_set_lda['Label'].values)
end_train_lda = time.time_ns()
time_train_lda = (end_train_lda-start_train_lda)/1000000000
print("lda train time:{}s".format(time_train_lda))

start_test_lda = time.time_ns()
test_lda = lda.transform(y_std_lda)
end_test_lda = time.time_ns()
time_test_lda = (end_test_lda-start_test_lda)/1000000000
print("lda train time:{}s".format(time_test_lda))
```

– KNN

```
k = [1,5,10,15,20,25,30,35]
for i in range(len(k)):
    #fit knn
    neigh = KNeighborsClassifier(n_neighbors=k[i])
    start_train_knn = time.time_ns()
    neigh.fit(train, train_set['Label'].values)
    end_train_knn = time.time_ns()
    time_train_knn = (end_train_knn-start_train_knn)/1000000000
    print("knn train time:{}s".format(time_train_knn))
    #get score
    scores = neigh.score(train, train_set['Label'].values)
    print("rt score:{}".format(scores))
    start_test_knn = time.time_ns()
    #predict test result
    result_knn = neigh.predict(test)
    end_test_knn = time.time_ns()
    time_test_knn = (end_test_knn-start_test_knn)/1000000000
    print("knn test time:{}s".format(time_test_knn))
    #save as csv
    dataframe = pd.DataFrame({'Label':result_knn})
    dataframe.to_csv("result{num}.csv".format(num = i),index=False,sep=',')
```

Figure 9: KNN

– Random Forest

```
from sklearn.ensemble import RandomForestClassifier
xLabel = [5,10,50,150,200]
yLabel = [3,5,10,None]
for i in range(len(xLabel)):
    temp = []
    for j in range(len(yLabel)):
        #fit random forest
        clf_rt = RandomForestClassifier(n_estimators=xLabel[i],max_depth=yLabel[j],random_state=42)
        start_train_rt = time.time_ns()
        clf_rt.fit(train, train_set['Label'].values)
        end_train_rt = time.time_ns()
        time_train_rt = (end_train_rt-start_train_rt)/1000000000
        print("rt train time:{}s".format(time_train_rt))
        #get score
        scores = cross_val_score(clf_rt,train, train_set['Label'].values,cv=10)
        print("rt score:{}".format(scores.mean()))
        start_test_rt = time.time_ns()
        #predict test
        result_rt = clf_rt.predict(test)
        end_test_rt = time.time_ns()
        time_test_rt = (end_test_rt-start_test_rt)/1000000000
        print("rt test time:{}s".format(time_test_rt))
        #save test result
        dataframe = pd.DataFrame({'Label':result_rt})
        dataframe.to_csv("result{numi}{numj}.csv".format(numi = i,numj = j),index=False,sep=',')
```

Figure 10: Random Forest

– SVC

```
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
tempC = [0.1,0.5,1,2,5,10,20,50]
for i in range(len(tempC)):
    #fit svc
    clf_svc = SVC(C= tempC[i],random_state=42)
    start_train_svc = time.time_ns()
    clf_svc.fit(train, train_set['Label'].values)
    end_train_svc = time.time_ns()
    time_train_svc = (end_train_svc-start_train_svc)/1000000000
    print("svc train time:{}s".format(time_train_svc))
    #get score
    scores = cross_val_score(clf_svc,train, train_set['Label'].values,cv=10)
    print("svc score:{}".format(scores.mean()))
    start_test_svc = time.time_ns()
    #predict test
    result_svc = clf_svc.predict(test)
    end_test_svc = time.time_ns()
    time_test_svc = (end_test_svc-start_test_svc)/1000000000
    print("svc test time:{}s".format(time_test_svc))
    #save test result
    dataframe = pd.DataFrame({'Label':result_svc})
    dataframe.to_csv("result_svc{numi}.csv".format(numi = i),index=False,sep=',')
```

Figure 11: SVC

Within SVC, we used its RBF algorithm and mainly performed changes on parameter c and gamma.

```
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
tempC = [0.1,0.5,1,2,5]
gamma = [0.1,0.05,0.005,0.001]
for i in range(len(tempC)):
    for j in range(len(gamma)):
        #add change parameter gamma into the code
        clf_svc = SVC(C= tempC[i],gamma = gamma[j],random_state=42)
        start_train_svc = time.time_ns()
        clf_svc.fit(train, train_set['Label'].values)
        end_train_svc = time.time_ns()
        time_train_svc = (end_train_svc-start_train_svc)/1000000000
        print("svc train time:{}s".format(time_train_svc))
        #get score
        scores = cross_val_score(clf_svc,train, train_set['Label'].values,cv=10)
        print("svc score:{}".format(scores.mean()))
        start_test_svc = time.time_ns()
        #predict the result
        result_svc = clf_svc.predict(test)
        end_test_svc = time.time_ns()
        time_test_svc = (end_test_svc-start_test_svc)/1000000000
        print("svc test time:{}s".format(time_test_svc))
        dataframe = pd.DataFrame({'Label':result_svc})
        dataframe.to_csv("20result_svc{numi}_{numj}.csv".format(numi = i,numj = j),index=False,sep=',')
```

Figure 12: RBF

– XGBoost

```
param_list = [("eta", 0.08),
              ("max_depth", 7),
              ("subsample", 0.8),
              ("colsample_bytree", 0.8),
              ("objective", "multi:softmax"),
              ("eval_metric", "merror"),
              ("alpha", 2), ("lambda", 1),
              ("num_class", 5)]
n_rounds = 150
early_stopping = 20

new_train = xgb.DMatrix(train_data_withoutLabel, label=labels)
new_val = xgb.DMatrix(X_test, label=y_test)
eval_list = [(new_train, "train"), (new_val, "validation")]

xg_reg = xgb.train(param_list,
                   new_train,
                   n_rounds,
                   evals=eval_list,
                   early_stopping_rounds=early_stopping,
                   verbose_eval=True)
```

Figure 13: XGBoost:Train

```
start = time.time()
xgb_clf2 = XGBClassifier(n_estimators=500, n_jobs=-1, learning_rate=0.5, seed=0)
xgb_clf2.fit(sec_X_train,sec_y_train)
end = time.time()
print("Total Calculation time: {} sec".format((round(end - start, 3))))

y_pred = xgb_clf2.predict(sec_X_test)
print(accuracy_score(sec_y_test, y_pred))
```

Figure 14: XGBoost:XGBClassifier

- 1.3 Kaggle Competition Score 89.7
- 1.4 Result Analysis
  - Run time performance for training and testing.
    - Feature extraction
      The comparison of run time performance on feature extraction is done between the combination of PCA + SVM and LDA + SVM. As shown in Figures 8 and 9, SVM performs on data extracted by PCA consumes longer training and testing time than data extracted by LDA.

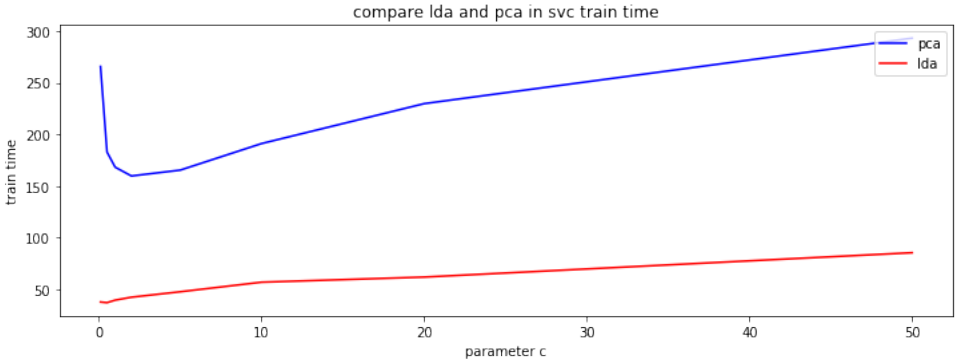


Figure 15: Train Time

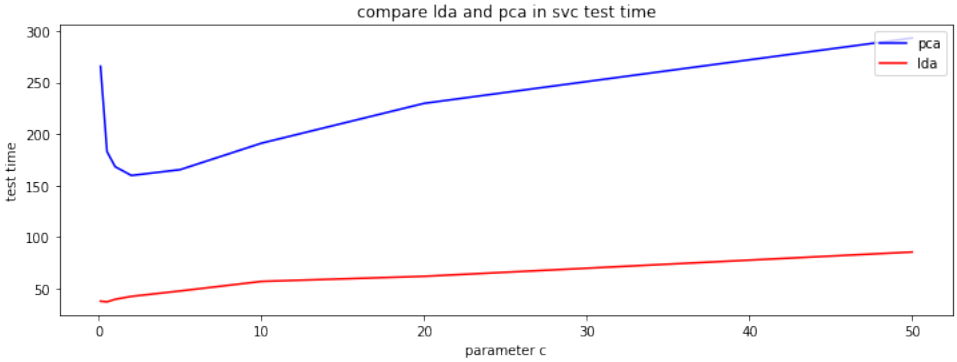


Figure 16: Test time

* Classifier XGBoost
  As an improved Classifier extended from Gradient Boost, the time efficiency is greatly increased. It helped our group to finish the test we weren't able to finish with the previous Gradient Boost. However, the overall accuracy didn't go as well as the assumption. Doing GridSeachCV on XGBoost also takes a tremendous amount of time which didn't lead to any result. For any parameters that we turned, please refer to the code blocks in section Implementation.
* Classifier KNN vs Random Forest
  As shown in figure 10 to 13, the train time and test time using KNN inverse with Random Forest. Training using KNN is faster, but the testing takes longer. For the same way around, Random Forest seems to spend more time on training, but the testing process is faster.

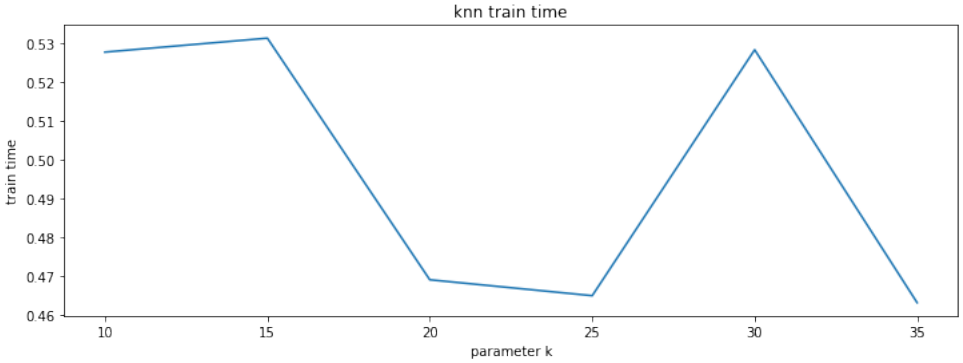


Figure 17: Train Time

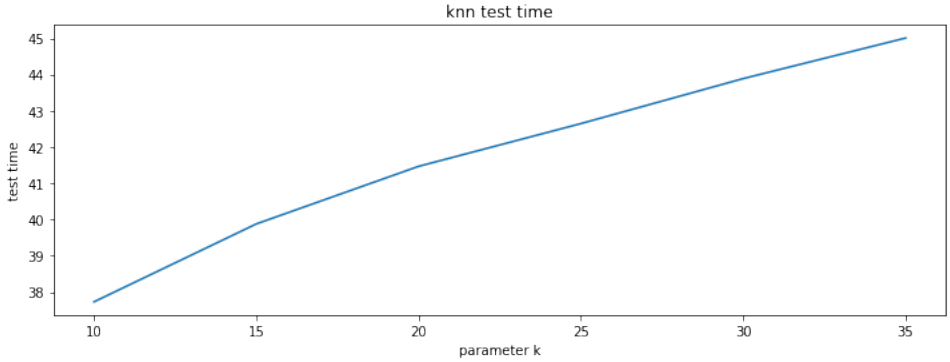


Figure 18: Train Time

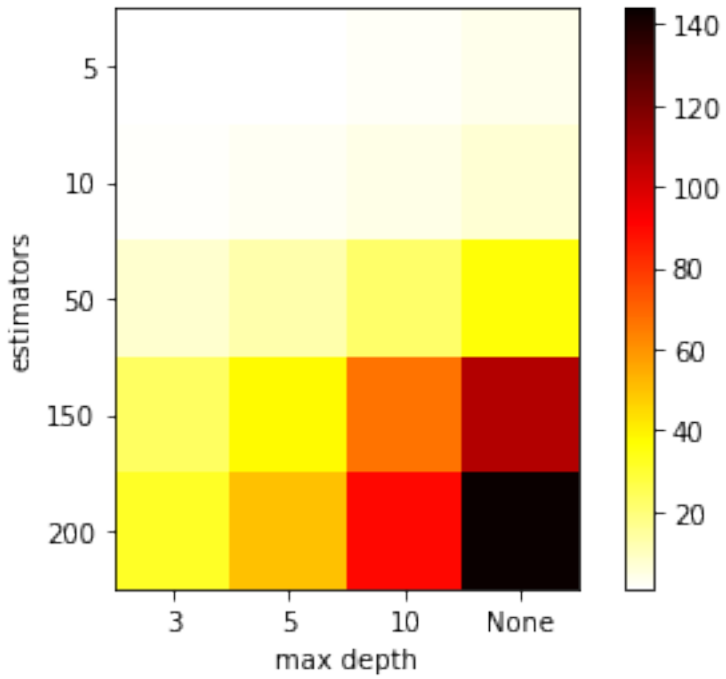


Figure 19: Train Time

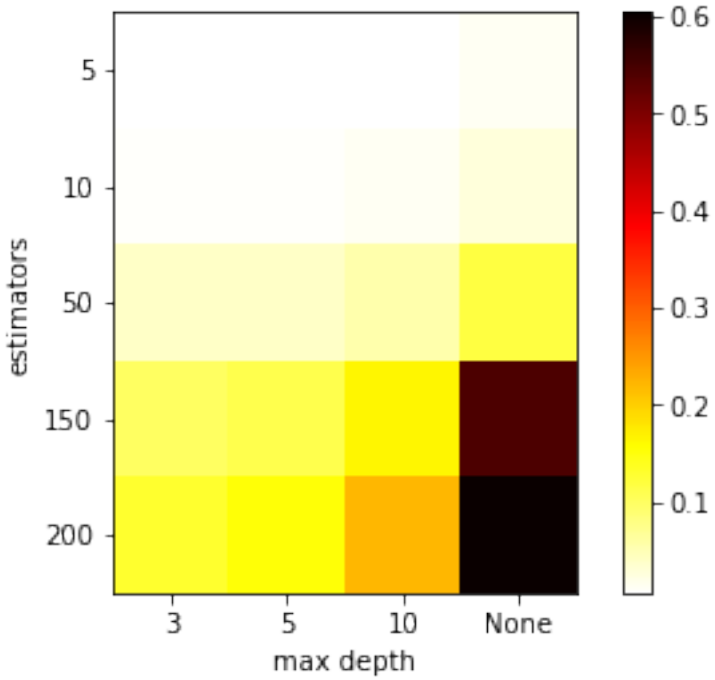


Figure 20: Test Time

* Classifier SVM
  The run time performance on SVM is done between two non-linear SVM algorithm, Polynomial, and RBF. The results for these two methods are close to each other.
  Overall, all methods share the same characteristic that the more complex the parameters are used, the longer it consume on training.

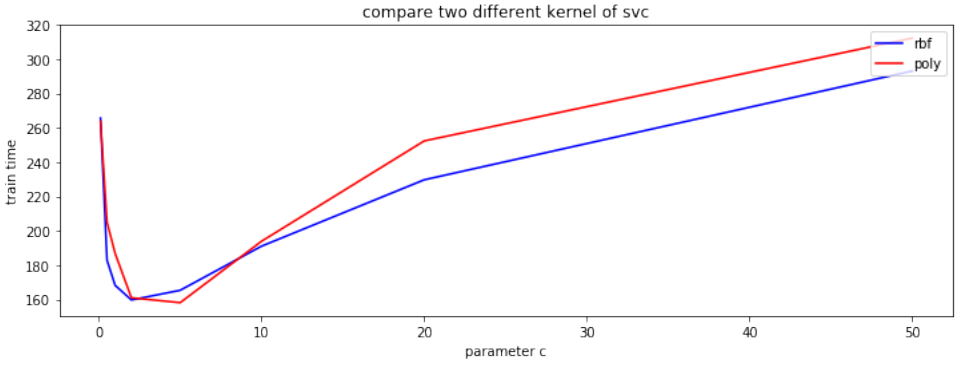


Figure 21: Train Time

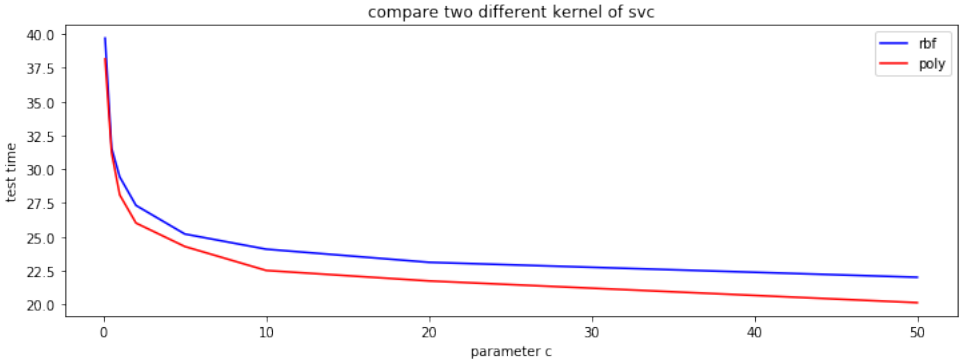


Figure 22: Test Time

* RBF with different principal components.
  On average, the number of principal components kept from PCA, the longer training and testing time consumed by RBF.

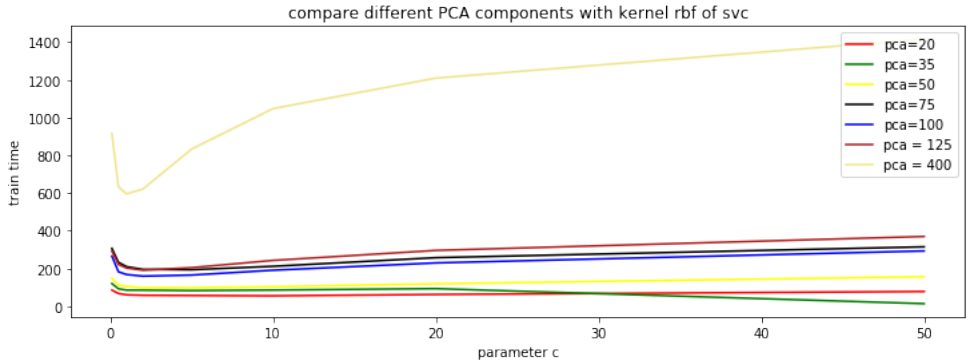


Figure 23: Test Time

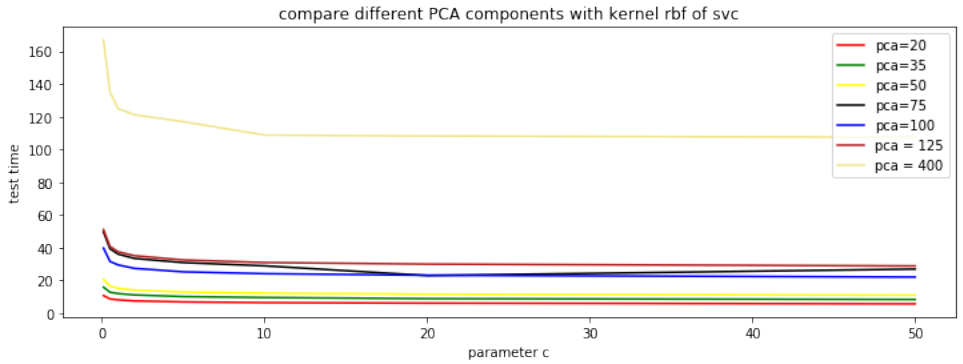


Figure 24: Test Time

- Comparison of the different algorithms and parameters you tried.
  - KNN vs. Random Forest Same to the comparison result in the previous time section, the accuracy scores of KNN and Random Forest with the effect of different parameters are inverse to each other. In KNN, large the parameter leads to underfit. On the other hand, larger parameters for Random Forest leads to higher accuracy score. Overall, KNN trains a more accurate model than Random Forest in this experiment.

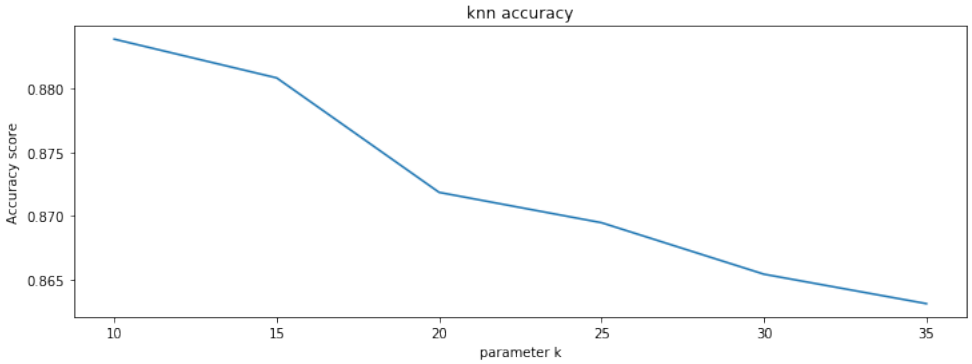


Figure 25: Accuracy Score

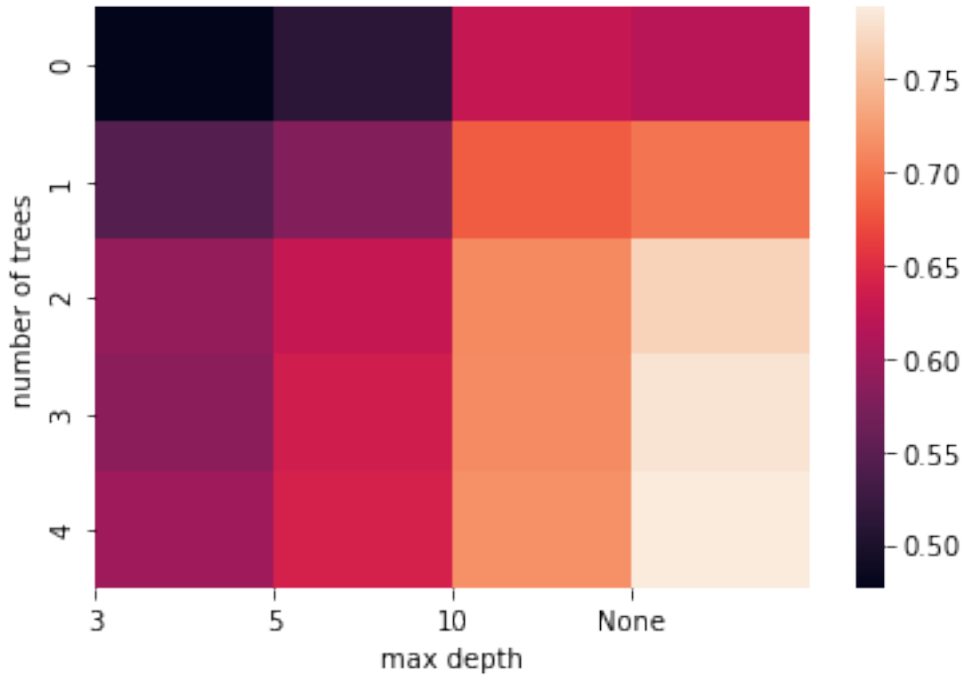


Figure 26: Accuracy Score

* SVM Poly vs. SVM RBF As shown in SVM training, RBF trains better model than Poly.

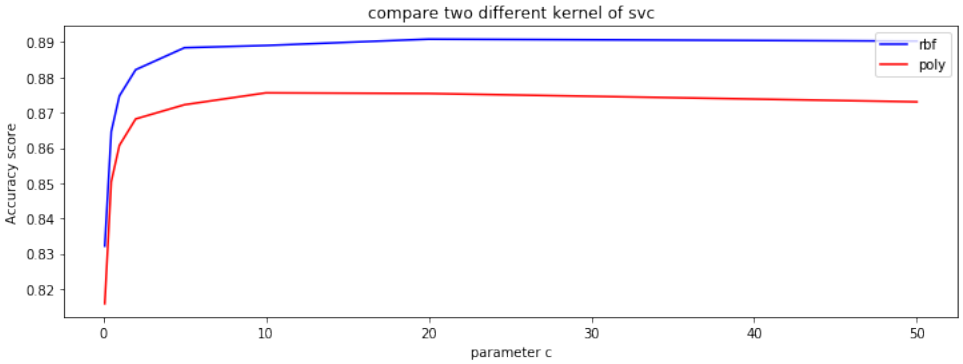


Figure 27: Accuracy Score

* PCA + RBF vs. LDA + RBF The result in the comparison proves the assumption that PCA is a better extraction method than LDA for the special relationships between each class with the data set.

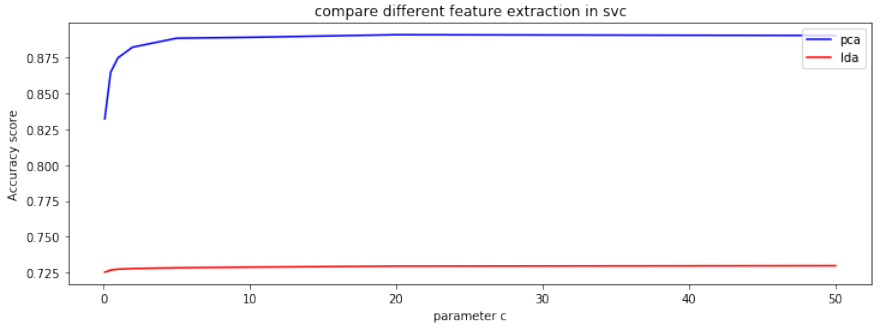


Figure 28: Accuracy Score

* RBF + PCA with different principle components Base on the graph, larger c in RBF seems to close the gap between a different number of principle components. Small PC (25, 35) may underfit the model and large PC (400) may overfit the model. A relatively similar and optimal range of PCS is from 50 to 125. And see the mean 75 as the most suitable selection.

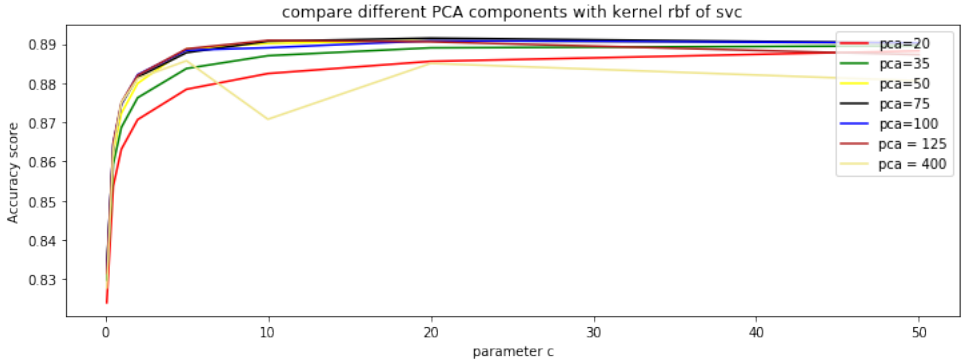


Figure 29: Accuracy Score

* Parameter gamma and c for SVM Using PCA 25 and 35 as example, we can see that a smaller gamma and a larger c can improve the prediction accuracy. So far the best pair is (0.005, 5)

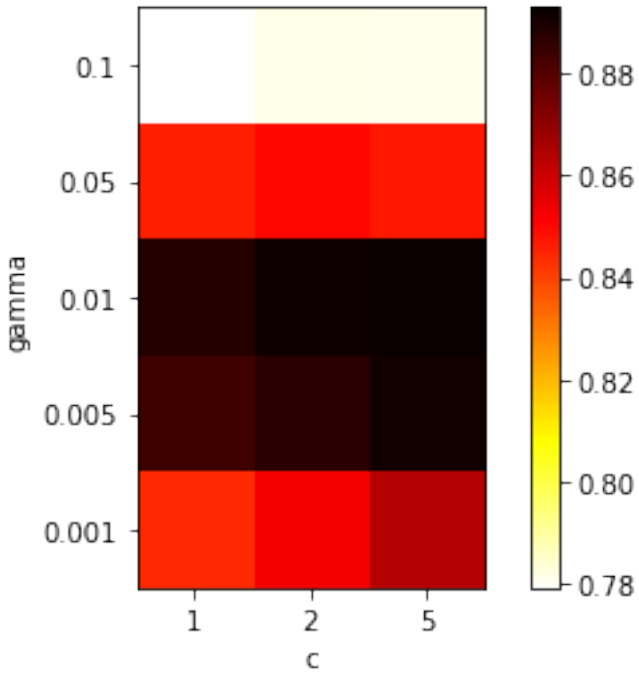


Figure 30: PCA 25

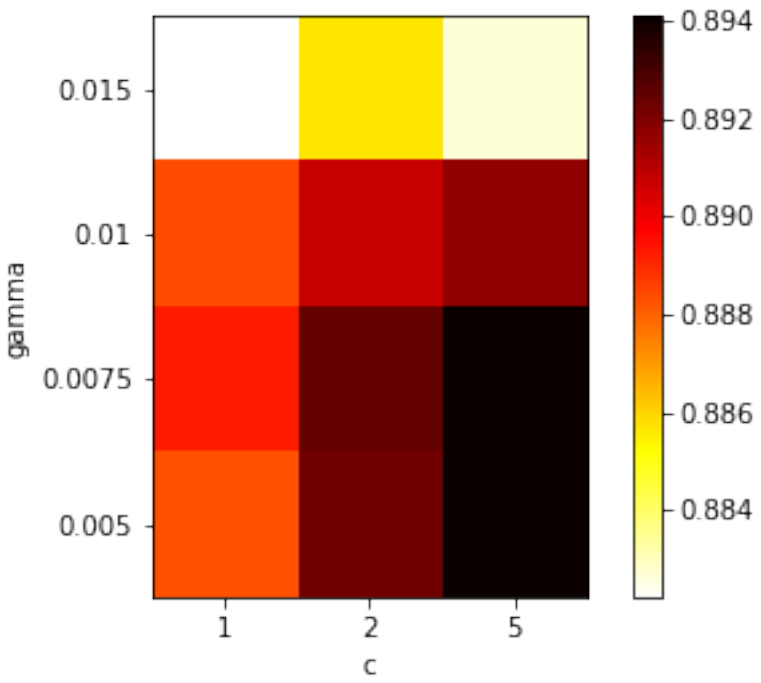


Figure 31: PCA 35

– Explanation of your model (algorithms, design choice, number of parameters). Our initial design choice includes PCA and LDA as feature extraction methods, SVM, Random Forest, and KNN as classifiers. After the initial round of tests, we found PCA + SVM is the best combination that provides an accurate prediction within limits resource frame. For PCA, we have a set of parameters from 25 to 400, and we choose 75 as the optimal number. For RBF, the c parameter list is from 1 to 5 and gamma is from 0.1 to 0.001. Furthermore, we organize different combinations of these parameters looking at the best model.
– Use a ROC curve used for some method in your initial or results analysis such as exploring the impact on the accuracy of some parameters.

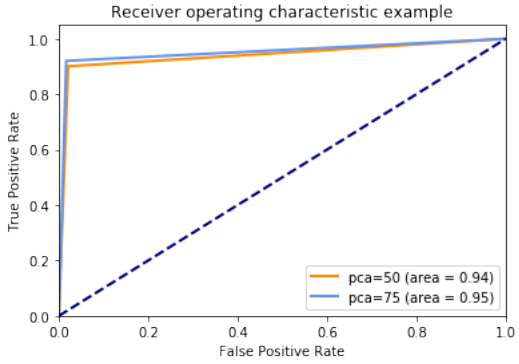


Figure 32: Different PCA components

For this ROC curve plot, SVM+PCA(component=50) and SVM+PCA(component=75) two methods have been used to train the model and plot the ROC curve. From the plot, when PCA(component=75), the AUC is 0.95, while PCA(component=50), the AUC is 0.94. Therefore, the model SVM+PCA(component=75) is a little bit better than SVM+PCA(component=50)

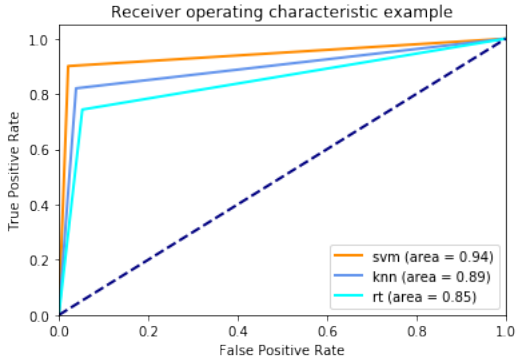


Figure 33: Different training method

For this ROC curve plot, different training methods have been used, which are KNN, Random Forest and SVM. From the plot, the AUC of SVM is the largest, which means than SVM is the better training method for the data.

– Evaluate your code with other metrics on the training data (by using some of it as test data) and argue for the benefit of your approach.

	precision	recall	f1-score	support
Class 0	0.94	0.95	0.95	2379
Class 1	0.87	0.88	0.88	2424
Class 2	0.85	0.83	0.84	2364
Class 3	0.86	0.86	0.86	2367
Class 4	0.93	0.92	0.93	2466
accuracy			0.89	12000
macro avg	0.89	0.89	0.89	12000
weighted avg	0.89	0.89	0.89	12000

Figure 34: PCA 25

The first metric we use is F1-score, recall, and precision.
$Precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$
Precision talks about how precise our model is out of those predicted positive, how many of them are positive. So precision can use to determine whether the cost of FP is high.

$Recall = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$
Recall actually calculates how many of the Actual Positives our model capture through labeling it as TP. So Recall can be used to determine whether the model has a high cost associated with FN.
$F1Score = \frac{2*Precision*Recall}{Precision+Recall}$
F1-score can be used to seek a balance between Precision and Recall.
For our model, the average of Precision and Recall and f1-score are both 0.89, which a relatively good score. Therefore, our approach is appropriate.

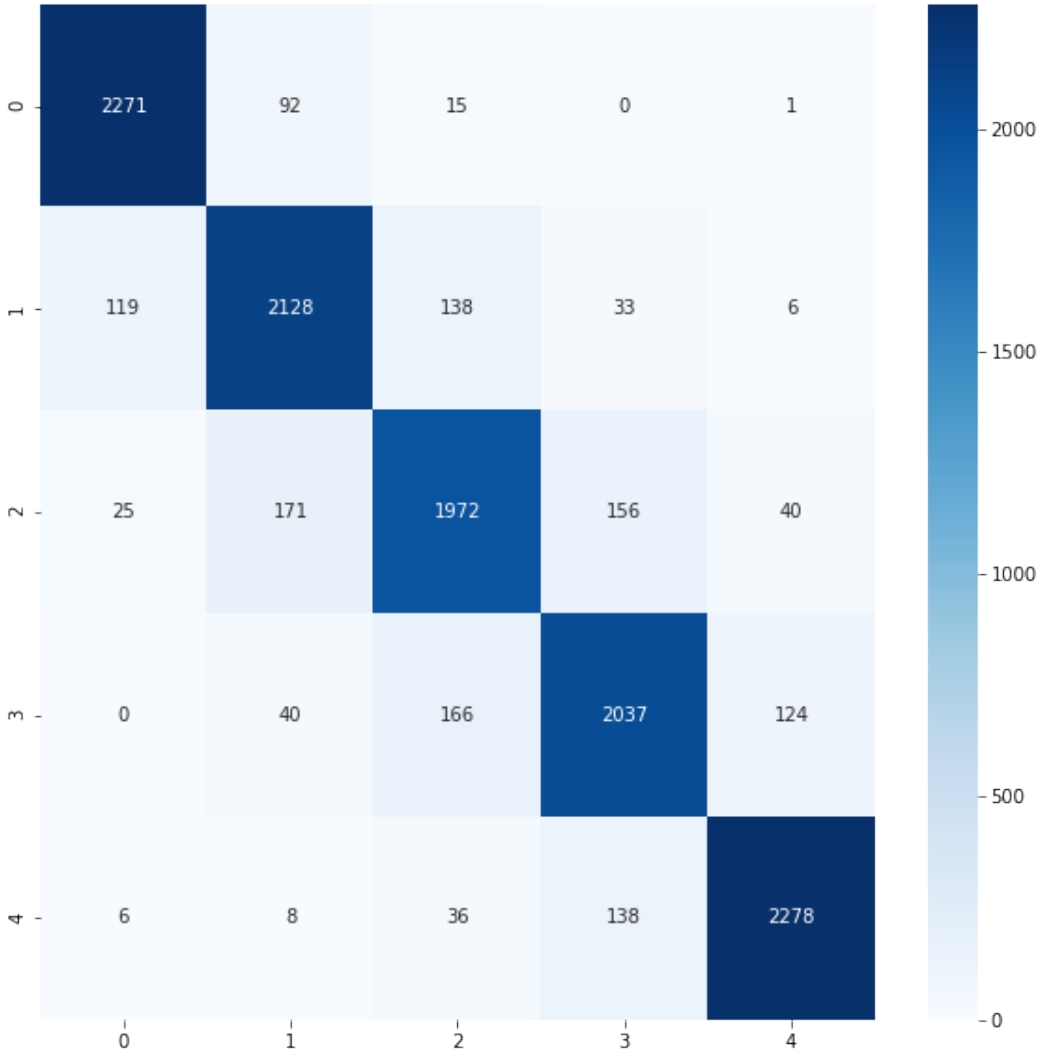


Figure 35: PCA 25

Besides, a visualization derived from the confusion matrix can be shown in the form of a heatmap, which displays in the distribution of the actual label of each false positive and true negative from the prior confusion matrix. Labels in class 0 and 5 are more correctly predicted compare to class 2,3, and 4. By the assumption, class 0 to 5 respectively represent a level of greyscale from white to black. Thus, it is coherent with the result in the confusion matrix for that adjacent classes are similar to each other than edge class, 0 and 4 should be discernible in the total dataset.

2. Question 2: Classification: Convolutional Neural Networks

   - 2.1: Design and Implementation Choices of your Model

     We use Keras and Tensor Flow to build CNN models. For all the model we use Adam optimizer to reach the global minimal while training out model. And use Checkpoint to record the best score model. For each model, the code architecture is demonstrated below.

     For this assignment, we choose to build four CNN models which include 2 sequential models, a LeNet5 model, and a VGG16 model.

     - Sequential model 1
       - 1 x convolution layer of 32 channel of 3x3 kernel
       - 1 x maxpool layer of 2x2 pool size
       - 1 x dropout rate 0.25
       - 1 x convolution layer of 64 channel of 3x3 kernel
       - 1 x maxpool layer of 2x2 pool size
       - 1 x dropout rate 0.25
       - 1 x convolution layer of 128 channel of 3x3 kernel
       - 1 x dropout rate 0.4
       - 1 x Dense layer of 128 units
       - 1 x Dense Softmax layer of 5 units
     - Sequential Model 2
       - 1 x convolution layer of 32 channel of 3x3 kernel
       - 1 x batch normalization
       - 1 x convolution layer of 32 channel of 3x3 kernel
       - 1 x batch normalization
       - 1 x maxpool layer of 2x2 pool size
       - 1 x dropout rate 0.25
       - 1 x convolution layer of 64 channel of 3x3 kernel
       - 1 x batch normalization
       - 1 x dropout rate 0.25
       - 3 x convolution layer of 256 channel of 3x3 kernel
       - 1 x batch normalization
       - 1 x maxpool layer of 2x2 pool size
       - 1 x dropout rate 0.25
       - 1 x Dense layer of 512 units
       - 1 x batch normalization
       - 1 x dropout rate 0.5
       - 1 x Dense layer of 128 units
       - 1 x batch normalization
       - 1 x dropout rate 0.5
       - 1 x Dense Softmax layer of 5 units
     - LeNet5

    - The LeNet-5 architecture consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier.
    - 1 x convolution layer of 6 channel of 5x5 kernel
    - 1 x average pool layer of 2x2 pool size
    - 1 x convolution layer of 15 channel of 5x5 kernel
    - 1 x average layer of 2x2 pool size
    - 1 x Dense layer of 84 units
    - 1 x Dense Softmax layer of 5 units
  - VGG16
    - 1 x convolution layer of 64 channel of 3x3 kernel and same padding
    - 1 x dropout rate 0.25
    - 1 x convolution layer of 64 channel of 3x3 kernel and same padding
    - 1 x dropout rate 0.4
    - 1 x maxpool layer of 2x2 pool size and stride 2x2
    - 1 x convolution layer of 128 channel of 3x3 kernel and same padding
    - 1 x dropout rate 0.25
    - 1 x convolution layer of 128 channel of 3x3 kernel and same padding
    - 1 x dropout rate 0.4
    - 1 x maxpool layer of 2x2 pool size and stride 2x2
    - 1 x convolution layer of 256 channel of 3x3 kernel and same padding
    - 1 x dropout rate 0.25
    - 1 x convolution layer of 256 channel of 3x3 kernel and same padding
    - 1 x dropout rate 0.4
    - 1 x convolution layer of 256 channel of 3x3 kernel and same padding
    - 1 x maxpool layer of 2x2 pool size and stride 2x2

- 2.3 Implementation of your Design Choices
  - Data process.
    - Reshape - In the first step, we collected data, and split it into train and validation. Then reshape both train and validation sets into a matrix of (28,28,1) as required by Keras.
    - Regularization - Since the pixel values are often stored as integers between 0 to 255, the range of a single 8-bit byte. To optimize run time efficiency, they are divide by 255, scaled down to [0,1]. Here, we achieve Zero mean and unit variance.
    - One hot encode - The labels are given as integers between 0 and 4. We need to one hot encode them , E.g. 2 [0, 0, 1, 0, 0]. We have 5 digits [0 - 4] or classes, therefore we one-hot-encode the target variable with 5 classes

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list the files in the input directory

from subprocess import check_output

from keras.utils import to_categorical
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from IPython.display import SVG
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
from keras.callbacks import ModelCheckpoint
```

```
#read the data
data_train = pd.read_csv('train.csv')
data_test = pd.read_csv('testX.csv')
df = data_train.copy()
#get the count of data for different label
print(df['Label'].value_counts())

img_rows, img_cols = 28, 28
input_shape = (img_rows,img_cols, 1)
#seperate data and label
X = np.array(data_train.iloc[:, 2:])
y = np.array(data_train.iloc[:, 1])

#split into train and validation
X_train, X_val, y_train_temp, y_val_temp = train_test_split(X, y, test_size=0.2, random_state=13)

#get category of label
y_train = to_categorical(y_train_temp)
y_val = to_categorical(y_val_temp)
#Test data
X_test = np.array(data_test.iloc[:, 1:])
```

```
#process the data
X_train = X_train.reshape(X_train.shape[0],img_rows,img_cols, 1)
X_test = X_test.reshape(X_test.shape[0],img_rows,img_cols, 1)
X_val = X_val.reshape(X_val.shape[0],img_rows,img_cols, 1)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_val = X_val.astype('float32')
X_train /= 255
X_test /= 255
X_val /= 255
```

– For model 1, firstly we use Sequential() to build the model. Next we choose Adam as optimizer and Categorical_crossentropy for loss function to compile the model, then we use ModelCheckPoint() to save the best result from the total epoches.

```
#model 1
batch_size = 256
num_classes = 5
epochs = 50

#input image dimensions
img_rows, img_cols = 28, 28

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 kernel_initializer='he_normal',
                 input_shape=input_shape))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(Dropout(0.4))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

```
#use Adam to train the model
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999 )
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=optimizer,
              metrics=['accuracy'])
model.summary()
#use checkpoint to save the best
filepath="mod2weights.best.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')
callbacks_list = [checkpoint]
history = model.fit(X_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,callbacks = callbacks_list,
          validation_data=(X_val, y_val))
```

– For model 2, firstly we use Sequential() to build the model, adding the layer as for LeNet5. Refer to model 1 for more detail.

```
#model 2
#lenet5
batch_size = 256
num_classes = 5
epochs = 100

model = Sequential()

model.add(layers.Conv2D(6, kernel_size=(5, 5), activation='relu', input_shape=(28,28,1)))

model.add(layers.AveragePooling2D(pool_size=(2, 2)))

model.add(layers.Conv2D(16, kernel_size=(5, 5), activation='relu'))

model.add(layers.AveragePooling2D(pool_size=(2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(84, activation='relu'))
model.add(layers.Dense(5, activation='softmax'))
```

```
#use Adma to train the model
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adam(lr=0.001 ), metrics=["accurac

#use checkpoint to save the best
filepath="mod4weights.best.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')
callbacks_list = [checkpoint]

history = model.fit(X_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,callbacks=callbacks_list,
          validation_data=(X_val, y_val))
```

– For model 3, the process is identical to model 1

```
batch_size = 128
num_classes = 5
epochs = 50

cnn4 = Sequential()
cnn4.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
cnn4.add(BatchNormalization())

cnn4.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
cnn4.add(BatchNormalization())
cnn4.add(MaxPooling2D(pool_size=(2, 2)))
cnn4.add(Dropout(0.25))

cnn4.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
cnn4.add(BatchNormalization())
cnn4.add(Dropout(0.25))

cnn4.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
cnn4.add(BatchNormalization())
cnn4.add(MaxPooling2D(pool_size=(2, 2)))
cnn4.add(Dropout(0.25))

cnn4.add(Flatten())

cnn4.add(Dense(512, activation='relu'))
cnn4.add(BatchNormalization())
cnn4.add(Dropout(0.5))

cnn4.add(Dense(128, activation='relu'))
cnn4.add(BatchNormalization())
cnn4.add(Dropout(0.5))
cnn4.add(Dense(num_classes, activation='softmax'))
```

```
#use adam to train the model
cnn4.compile(loss=keras.losses.categorical_crossentropy,
             optimizer=keras.optimizers.Adam(),
             metrics=['accuracy'])

#use checkpoint to save the best
filepath="mod5weights.best.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')
callbacks_list = [checkpoint]
history = cnn4.fit(X_train, y_train, batch_size = batch_size, epochs = epochs,
                           validation_data = (X_val, y_val), verbose=1,
                           callbacks = callbacks_list)
```

– For model 4, firstly we use Model() to build the model. The rest of the process is identical to model 1

```
input_tensor=Input(shape=X_train.shape[1:])
x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv1')(input_tensor)
x = Dropout(0.25)(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv2')(x)
x = Dropout(0.4)(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)
# Block 2
x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv1')(x)
x = Dropout(0.25)(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv2')(x)
x = Dropout(0.4)(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)
# Block 3
x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv1')(x)
x = Dropout(0.25)(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv2')(x)
x = Dropout(0.4)(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)
# Classification block
x = Flatten(name='flatten')(x)
x = Dense(1024, activation='relu', name='fc1')(x)
x = Dropout(0.5)(x)
x = Dense(1024, activation='relu', name='fc2')(x)
x = Dropout(0.5)(x)
x = Dense(5, activation='softmax', name='predictions')(x)
```

```
filepath="weights.best.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')
callbacks_list = [checkpoint]

model3 = Model(input_tensor, x)
model3.compile(loss = 'categorical_crossentropy', optimizer = "adam", metrics=['accuracy'])
model3.summary()
model3fit = model3.fit(X_train, y_train, validation_data=(X_val, y_val), batch_size=128, verbose=1, epochs=35,callbacks = callbacks_list)
```

- 2.4 Kaggle Competition Score 90.567
- 2.5 Results Analysis
  - Runtime performance for training and testing.
    By observation, LeNet, model 2, is the fastest CNN algorithm for it has only two convolutional layers. Simple CNN, model 1, is the second-fastest because it contains three convolutional layers. Model 3 is significantly slower than the previous 2 model for not only it has 4 layers, but also for each layer it requires a Batch Normalization. VGG16, model 4, is consuming the longest time to train because its algorithm is the most complex within these models.

Model	Time / EPOCH (sec)	Time/Step (us)
1	20	418
2	9	185
3	90	2000
4	1001	17000

  - Comparison of the different algorithms and parameters you tried.
    For each model, iteration starts with a batch size = 256 and a learning rate = 0.001. We train the model with these parameters for numerous times. From the result, we keep the best model and start the second iteration with a reduced learning rate. This process is repeated through several iterations until the accuracy reaches a staple level, which can not be further improved.

The chart below displays the finial accuracy score for each model.

Model	Accuracy				
	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
1	0.8808	0.89475	0.89758	0.9005	0.90142
2	0.88583	0.89033	0.89567	0.89692	0.89717
3	0.89767	0.90167	0.9045	0.9055	0.90625
4	0.89658	0.9055	0.9055	0.90725	*

Overall, the accuracy varies between +-6% in each model. The highest accuracy is retrieved from the model, but the trade-off is that time consumption is the longest. Thus, we weren't able to run the last iteration using model 4. Also, the accuracy increases for each iteration repetitively.

– Explanation of your model (algorithms, network architecture, optimizers, regularization, design choices, numbers of parameters)
For this project, we used two sequential models, a LeNet5 model, and a VGG16 model. Respectively, the network architecture flow diagram is shown in the figures below. Also may refer to section 2.1.

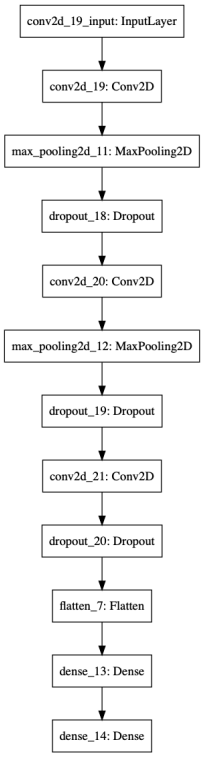


Figure 36: Sequential model 1

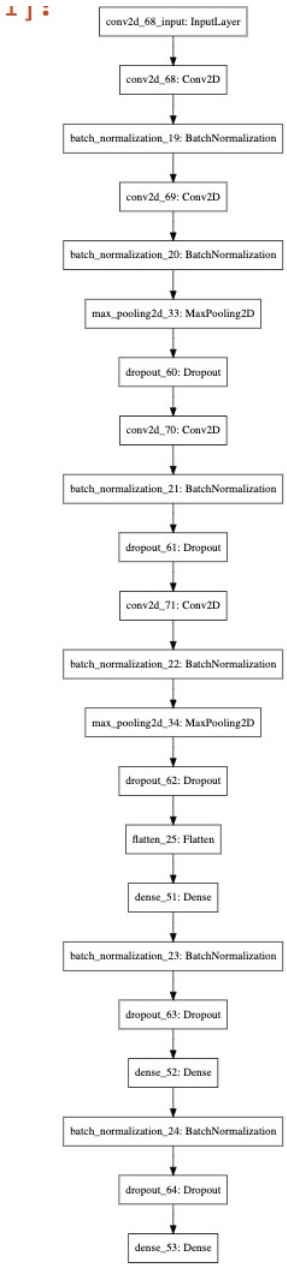


Figure 37: Sequential model 2

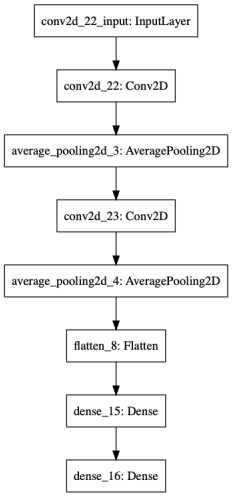


Figure 38: LeNet5 model

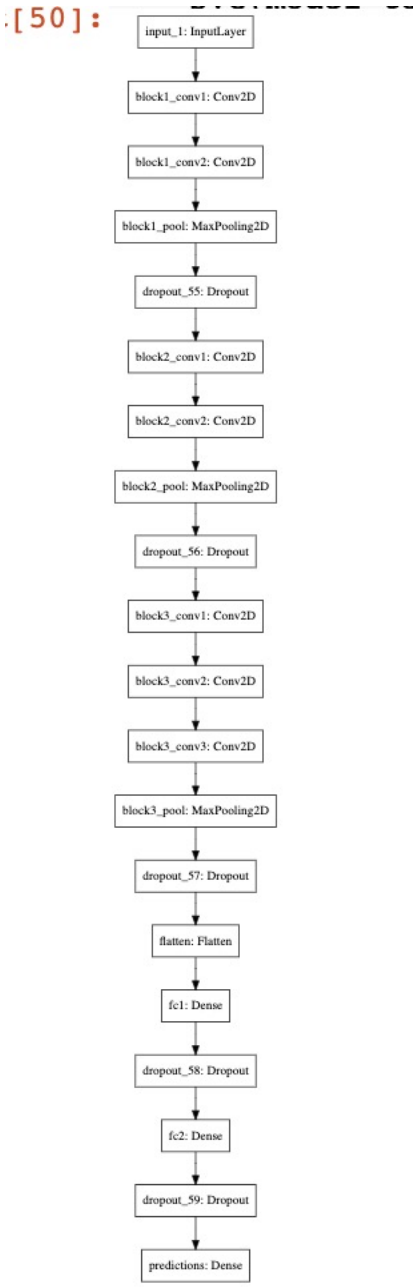


Figure 39: VGG16 model

The optimizer we used is Adam for all three models. Different from traditional optimizer that maintains a simple learning rate in scaling, Adam can adaptively update learning rate based on the method of moments in training. It can be considered as a combination of AdaGrad and RMSProp.(1)

Regulation is described in section 2.3 Data Process and design choice is described in section 2.1.
All of the tuned alternative parameters are shown in the figures below.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_31 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 13, 13, 32)        0
_________________________________________________________________
dropout_29 (Dropout)         (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
dropout_30 (Dropout)         (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 3, 3, 128)         73856
_________________________________________________________________
dropout_31 (Dropout)         (None, 3, 3, 128)         0
_________________________________________________________________
flatten_11 (Flatten)         (None, 1152)              0
_________________________________________________________________
dense_22 (Dense)             (None, 128)               147584
_________________________________________________________________
dense_23 (Dense)             (None, 5)                 645
=================================================================
Total params: 240,901
Trainable params: 240,901
Non-trainable params: 0
```

Figure 40: Sequential model 1

```
Model: "sequential_23"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_59 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
batch_normalization_13 (Batc (None, 26, 26, 32)        128
_________________________________________________________________
conv2d_60 (Conv2D)           (None, 24, 24, 32)        9248
_________________________________________________________________
batch_normalization_14 (Batc (None, 24, 24, 32)        128
_________________________________________________________________
max_pooling2d_29 (MaxPooling (None, 12, 12, 32)        0
_________________________________________________________________
dropout_47 (Dropout)         (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_61 (Conv2D)           (None, 10, 10, 64)        18496
_________________________________________________________________
batch_normalization_15 (Batc (None, 10, 10, 64)        256
_________________________________________________________________
dropout_48 (Dropout)         (None, 10, 10, 64)        0
_________________________________________________________________
conv2d_62 (Conv2D)           (None, 8, 8, 128)         73856
_________________________________________________________________
batch_normalization_16 (Batc (None, 8, 8, 128)         512
_________________________________________________________________
max_pooling2d_30 (MaxPooling (None, 4, 4, 128)         0
_________________________________________________________________
dropout_49 (Dropout)         (None, 4, 4, 128)         0
_________________________________________________________________
flatten_22 (Flatten)         (None, 2048)              0
_________________________________________________________________
dense_44 (Dense)             (None, 512)               1049088
_________________________________________________________________
batch_normalization_17 (Batc (None, 512)               2048
_________________________________________________________________
dropout_50 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_45 (Dense)             (None, 128)               65664
_________________________________________________________________
batch_normalization_18 (Batc (None, 128)               512
_________________________________________________________________
dropout_51 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_46 (Dense)             (None, 5)                 645
=================================================================
Total params: 1,220,901
Trainable params: 1,219,109
Non-trainable params: 1,792
```

Figure 41: Sequential model 2

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_34 (Conv2D)           (None, 24, 24, 6)         156
_________________________________________________________________
average_pooling2d_5 (Average (None, 12, 12, 6)         0
_________________________________________________________________
conv2d_35 (Conv2D)           (None, 8, 8, 16)          2416
_________________________________________________________________
average_pooling2d_6 (Average (None, 4, 4, 16)          0
_________________________________________________________________
flatten_12 (Flatten)         (None, 256)               0
_________________________________________________________________
dense_24 (Dense)             (None, 84)                21588
_________________________________________________________________
dense_25 (Dense)             (None, 5)                 425
=================================================================
Total params: 24,585
Trainable params: 24,585
Non-trainable params: 0
```

Figure 42: LeNet model 1

Model: "model_4"

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	(None, 28, 28, 1)	0
block1_conv1 (Conv2D)	(None, 28, 28, 64)	640
block1_conv2 (Conv2D)	(None, 28, 28, 64)	36928
block1_pool (MaxPooling2D)	(None, 14, 14, 64)	0
dropout_16 (Dropout)	(None, 14, 14, 64)	0
block2_conv1 (Conv2D)	(None, 14, 14, 128)	73856
block2_conv2 (Conv2D)	(None, 14, 14, 128)	147584
block2_pool (MaxPooling2D)	(None, 7, 7, 128)	0
dropout_17 (Dropout)	(None, 7, 7, 128)	0
block3_conv1 (Conv2D)	(None, 7, 7, 256)	295168
block3_conv2 (Conv2D)	(None, 7, 7, 256)	590080
block3_conv3 (Conv2D)	(None, 7, 7, 256)	590080
block3_pool (MaxPooling2D)	(None, 3, 3, 256)	0
dropout_18 (Dropout)	(None, 3, 3, 256)	0
flatten (Flatten)	(None, 2304)	0
fc1 (Dense)	(None, 1024)	2360320
dropout_19 (Dropout)	(None, 1024)	0
fc2 (Dense)	(None, 1024)	1049600
dropout_20 (Dropout)	(None, 1024)	0
predictions (Dense)	(None, 5)	5125

Total params: 5,149,381
Trainable params: 5,149,381
Non-trainable params: 0

Figure 43: VGG16

– You can use any plots to explain the performance of your approach. But at the very least produce two plots, one of training epoch vs. loss and one of classification accuracy vs. loss on both your training and test set.
We draw 6 comparison plots for Sequential mode 1, 2, and LeNet model. As for VGG16, time consumption exceeds the limitation we have available. Thus, it is excluded from this section.

  * Sequential mode 1

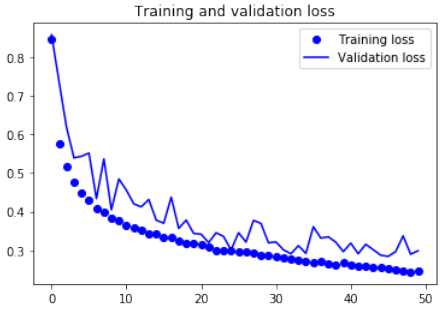


Figure 44: iteration 1

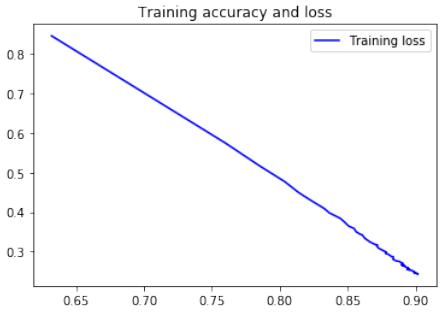


Figure 45: iteration 1

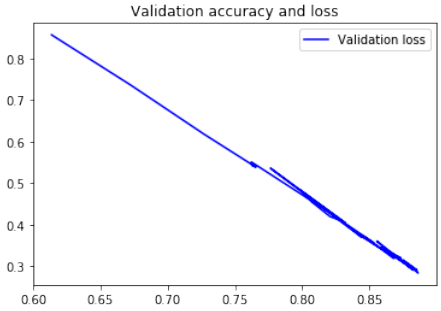


Figure 46: iteration 1

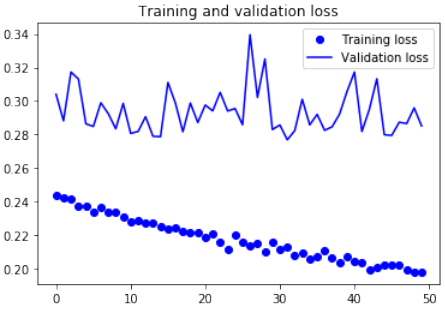


Figure 47: iteration 2

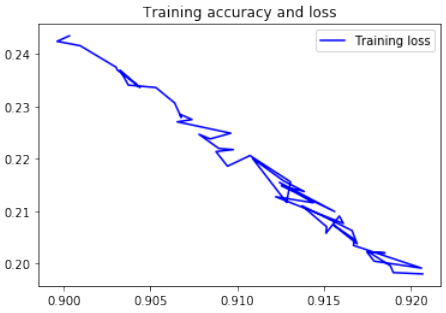


Figure 48: iteration 2

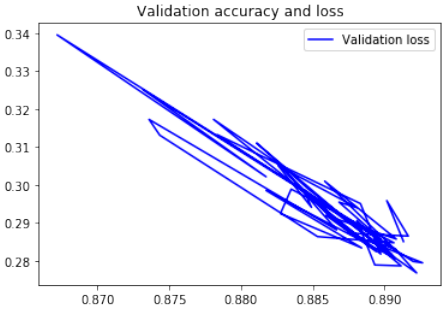


Figure 49: iteration 2

∗ Sequential model2

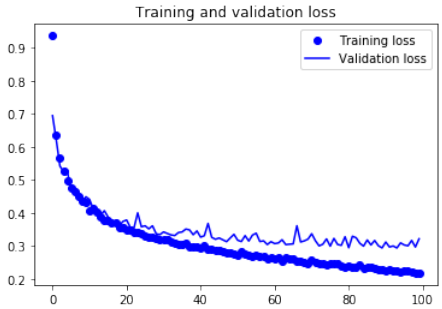


Figure 50: iteration 1

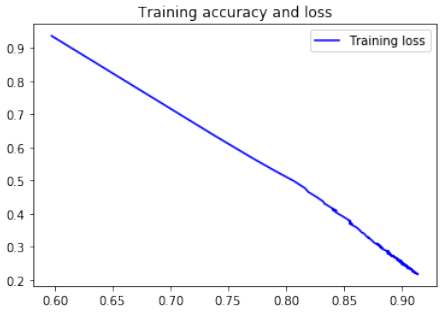


Figure 51: iteration 1

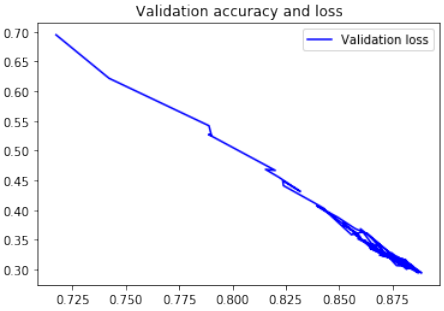


Figure 52: iteration 1

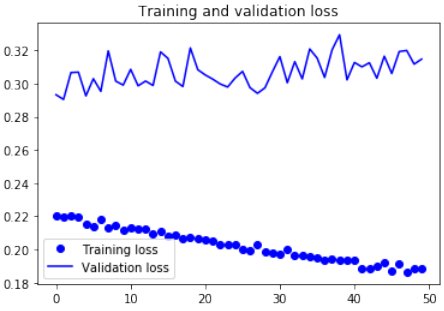


Figure 53: iteration 2

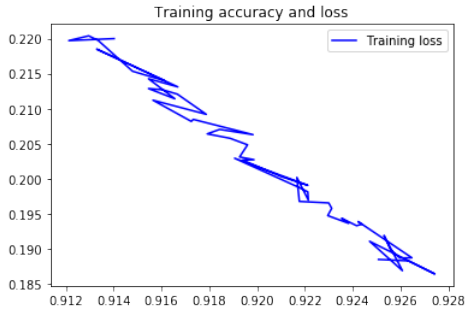


Figure 54: iteration 2

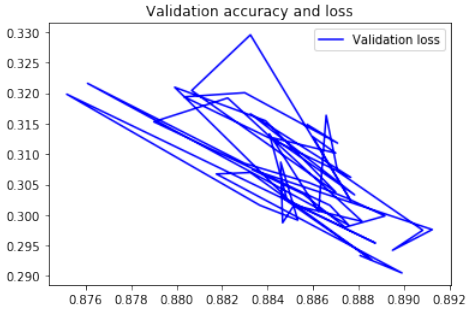


Figure 55: iteration 2

∗ LeNet5 model

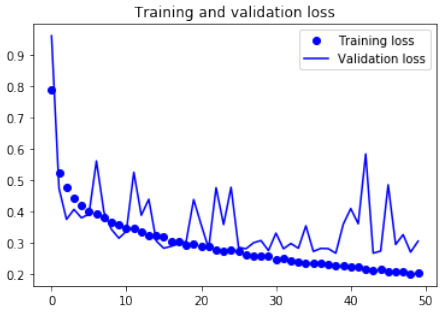


Figure 56: iteration 1

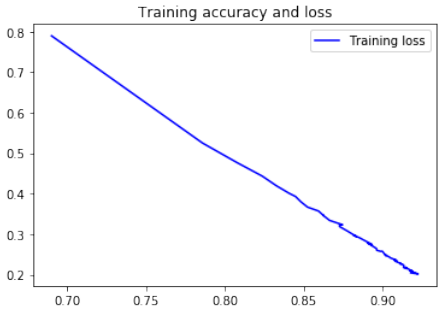


Figure 57: iteration 1

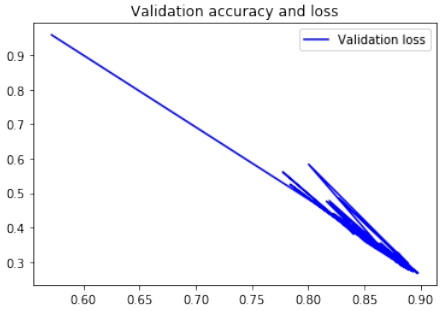


Figure 58: iteration 1

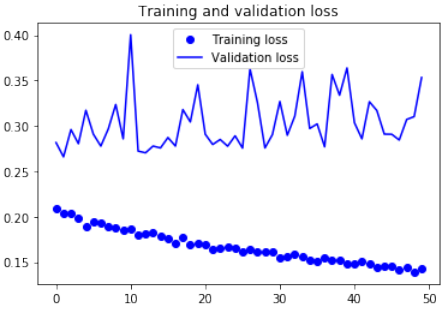


Figure 59: iteration 2

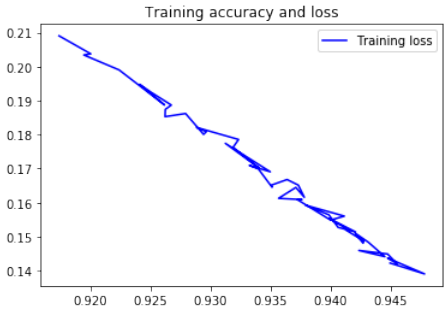


Figure 60: iteration 2

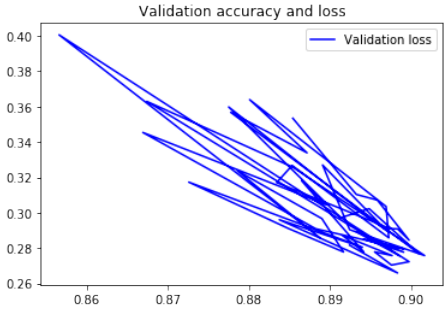


Figure 61: iteration 2

As talked in the previous section, new iterations further reduce learning rate and are build upon the epoch with the highest accuracy form previous iterations. The training and validation loss curves being close to each other in the first iteration, we can conclude that the Model is not overfitting the Data. Continue as iterations move on, both curve seems to move in a downward trend. The training loss curve seems to move smoothly while the validation loss curve contains multiple precipitous rises and falls. Two curves are further away from each other in the second iteration which means the increase of overfitting to the dataset. The overall trend for each iteration is similar to accuracy increase while the loss rate decrease. As iteration goes on, accuracy and loss plots start to lose their linear pattern, and the validation set suffers more from this syndrome than testing set

* ROC curve By observation, a similar performance is seem between three CNN models.

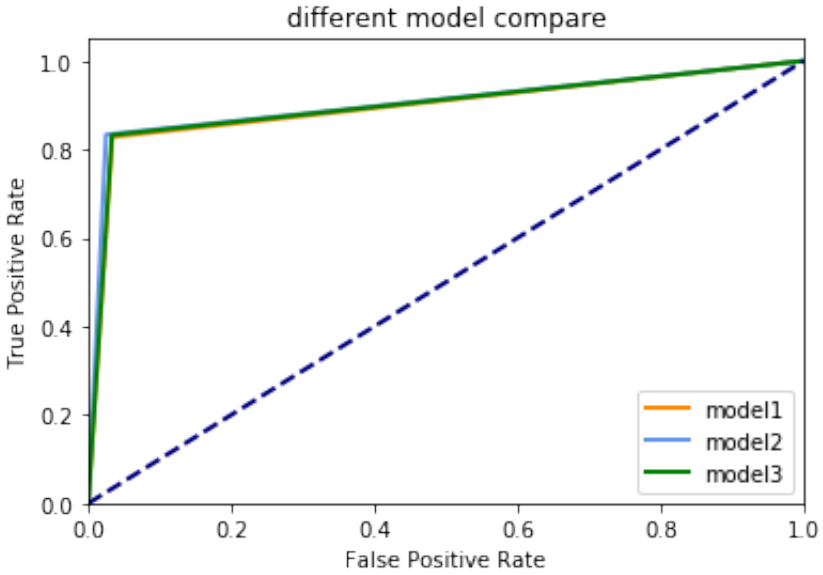


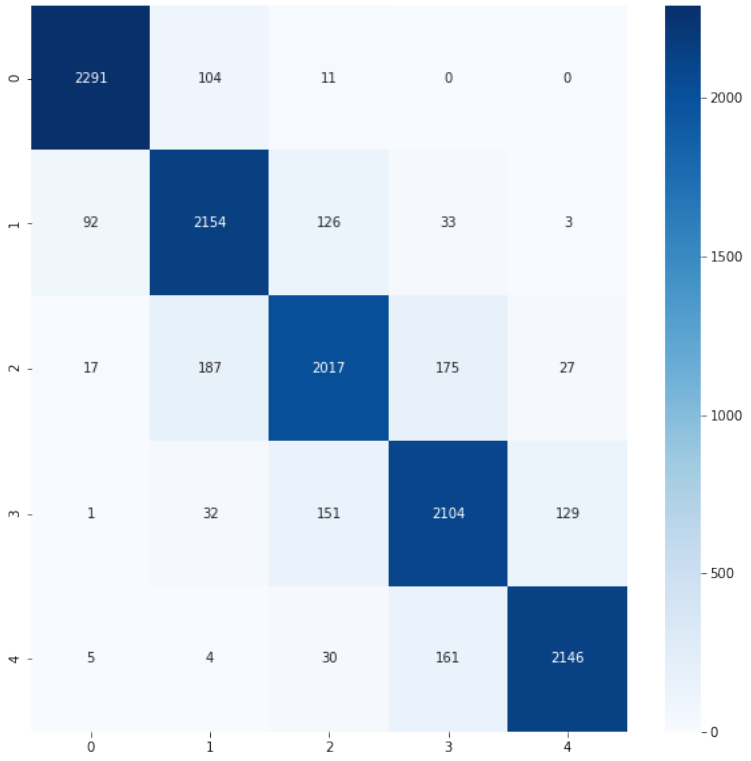
Figure 62: CNN ROC

– Evaluate your code with other metrics on the training data (by using some of it as test data) and argue for the benefit of your approach.
The first metric we use is F1-score, recall, and precision.
$Precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$
Precision talks about how precise our model is out of those predicted positive, how many of them are positive. So precision can use to determine whether the cost of FP is high.

$Recall = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$
Recall actually calculates how many of the Actual Positives our model capture through labeling it as TP. So Recall can be used to determine whether the model has a high cost associated with FN.
$F1Score = \frac{2 * Precision * Recall}{Precision + Recall}$
F1-score can be used to seek a balance between Precision and Recall.
For our models, the average of Precision and Recall and f1-score are both 0.91 for LeNet5, which a relative good score.
In addition, a visualization derived from the confusion matrix can be shown in form of a heatmap, which displays in the distribution of the actual label of each false positive and true negative from the prior confusion matrix. Labels in class 0 and 5 are more correctly predicted compare to class 2,3, and 4. By the assumption, class 0 to 5 respectively represent a level of greyscale from white to black. Thus, it is coherent with the result in the confusion matrix for that adjacent classes are similar to each other than edge class, 0 and 4 should be discernible in the total dataset.

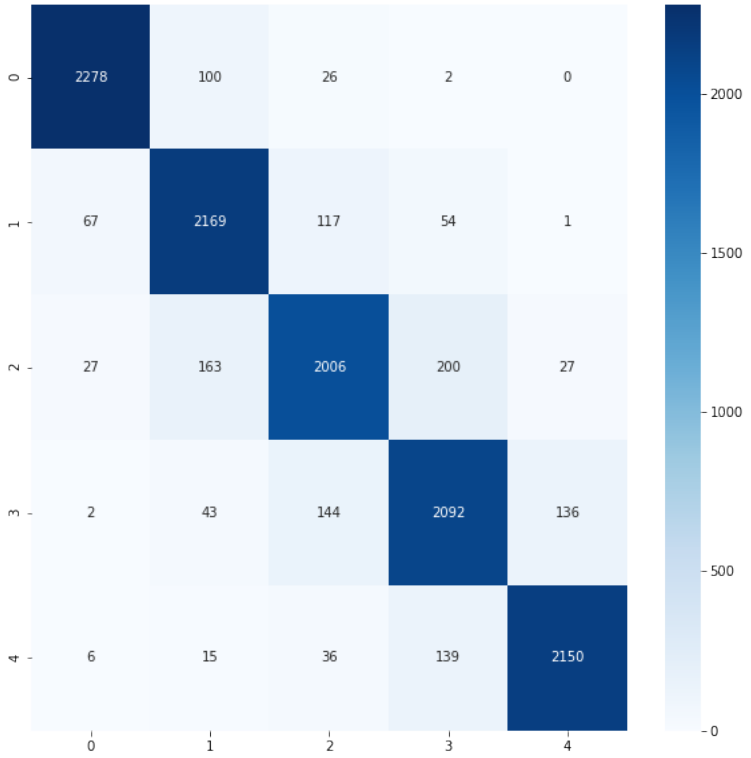
  * Sequential model 1

	precision	recall	f1-score	support
Class 0	0.96	0.95	0.95	2406
Class 1	0.87	0.90	0.89	2408
Class 2	0.86	0.83	0.84	2423
Class 3	0.84	0.87	0.85	2417
Class 4	0.93	0.92	0.92	2346
accuracy			0.89	12000
macro avg	0.89	0.89	0.89	12000
weighted avg	0.89	0.89	0.89	12000



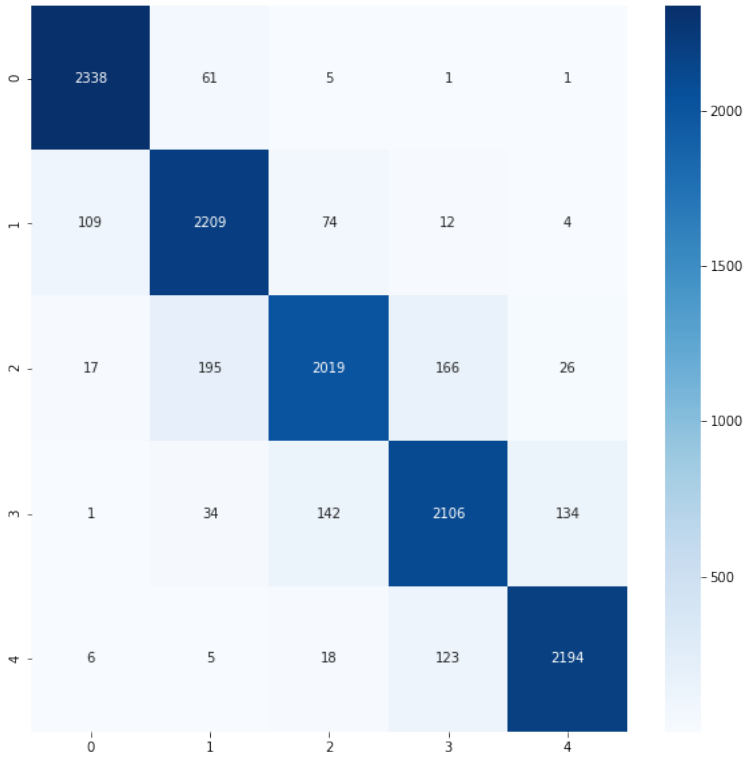
∗ Sequential model 2

	precision	recall	f1-score	support
Class 0	0.95	0.95	0.95	2406
Class 1	0.87	0.89	0.88	2408
Class 2	0.86	0.83	0.85	2423
Class 3	0.85	0.87	0.86	2417
Class 4	0.93	0.91	0.92	2346
accuracy			0.89	12000
macro avg	0.89	0.89	0.89	12000
weighted avg	0.89	0.89	0.89	12000



∗ LeNet 5

	precision	recall	f1-score	support
Class 0	0.95	0.97	0.96	2406
Class 1	0.88	0.92	0.90	2408
Class 2	0.89	0.83	0.86	2423
Class 3	0.87	0.87	0.87	2417
Class 4	0.93	0.94	0.93	2346
accuracy			0.91	12000
macro avg	0.91	0.91	0.91	12000
weighted avg	0.91	0.91	0.91	12000



1 reference

- $https://zhuanlan.zhihu.com/p/33385885$
- $https://xgboost.readthedocs.io/en/latest/$
- $https://engmrk.com/lenet-5-a-classic-cnn-architecture/$
- $https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c$
- $https://medium.com/@pechyonkin/key-deep-learning-architectures-lenet-5-6fc3c59e6f4$
- $https://classeval.wordpress.com/introduction/basic-evaluation-measures/$
- $https://blog.csdn.net/vesper305/article/details/44927047$
- $https://blog.csdn.net/qq_38410428/article/details/88106395$
- $https://blog.csdn.net/u013385925/article/details/80385873$
- $https://zhuanlan.zhihu.com/p/26293316$