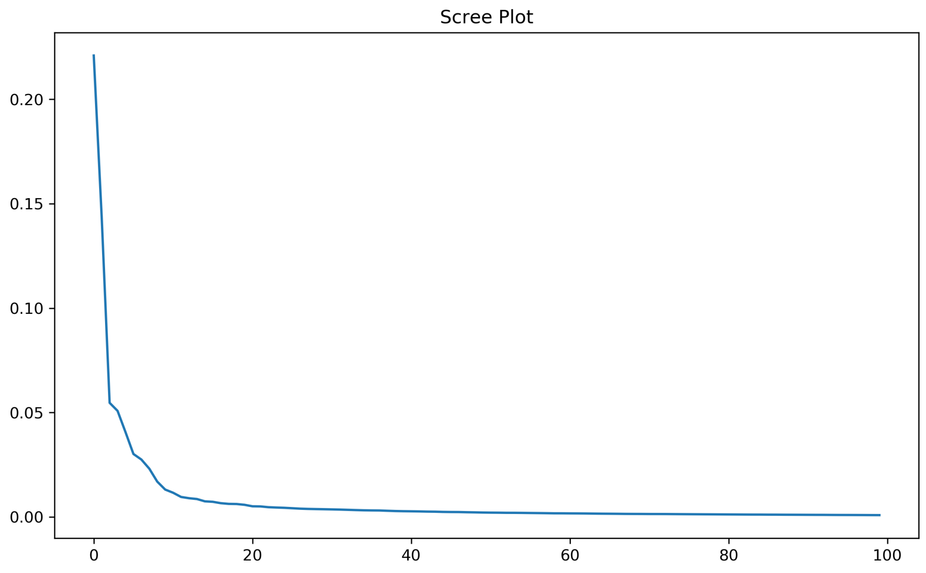
The source code is at the end of this document

1 Classification: Feature Extraction + Classical Methods

1.1 Explanation of Design and Implementation Choices of your Model

The models I used on Kaggle are XGBoost, SVM, Random Forest and Hist Gradient Boosting. Below is the procedure to show why I chose them.

Dataset was given with labels, so naturally supervised learning methods were considered. There are 784 dimensions in the dataset, therefore, PCA should be applied on it to extract the main features and to decrease the dimension. First I used standard scalar to centralize data and then drew the scree plot with top 100 principle components.



More than 99 percent of variance can be explained by the top 20 principle components. I took n_components = 30 for a better result.

The following classification methods with default parameters were applied on the PCA transformed data. (Except for MultinomialNB since Naive Bayes based on applying Bayes' theorem with strong independence assumptions between the features, and preprocessing will seriously impact the result.) Using 20 percent data as test set, their corresponding accuracy are shown below.

Classifier	**Accuracy**
MultiNomialNB	47.18%
Random Forest	86.95%
KNN	85.86%
SVM	86.13%
Logistic Regression	67.78%
LDA	63.68%
AdaBoost	61.03%
XGBoost	87.14%
Histogram-based Gradient Boosting	87.38%

I did not try Gradient Boosting because both XGBoost and Hist Gradient Boosting are significantly faster than it. E.g. In the Hist Gradient Boosting's User Guide

> These histogram-based estimators can be **orders of magnitude faster** than GradientBoostingClassifier and GradientBoostingRegressor when the number of samples is larger than tens of thousands of samples.

Random Forest, KNN, SVM, XGBoost and Hist Gradient Boosting were chosen for further parameter tuning. KNN's accuracy was always 1-2 percent lower than the others so I eliminated it as well. The working mechanism for the rest four classifiers are as follows:

Random Forest: uses a large number of uncorrelated individual decision trees, and each tree will give a class prediction. The class with the most votes will be the model's final prediction.

SVM: applies a kernel function on the data and finds hyperplanes to separate two classes, and picks the one that maximizes the margin. For multiclass problems like this one, the problem will be treated as multiple binary classifications(One-vs-All or One-vs-One).

Extreme Gradient Boosting: just like gradient boosting, xgboost builds multiple trees, and each tree is based on minimizing the loss of the previous tree. And the final prediction is based on the sum of the learning rate times the prediction of each tree. However, xgboost uses a different loss function and introduces other parameters to make the modeling more robust and faster.

Histogram-based Gradient Boosting: a new experimental implementation of gradient boosting trees in sklearn 0.21, no blog or paper currently explains its detailed implementation. According to documentation, it is inspired by LightGBM.

I chose the above algorithms because their performance is relatively good. And except for SVM, the other 3 are ensemble methods, which provide an extra degree of freedom in the classical bias/variance tradeoff and they are also unlikely to overfit. In addition, they are easy to run on multicores to save running time.

Apart from PCA alone as the data preprocessing method, I also tried histogram of oriented gradients(HOG) or HOG combined with PCA as data preprocessing according to this paper. HOG counts occurrences of gradient orientation in each localized area of an image, so that classification methods such as SVM can use these info to make prediction. However, the accuracy is only about 62%.

1.2 Implementation of your Design Choices

- Read data, apply standard scaler and PCA and train test split:

```
df = pd.read_csv('train.csv').drop(columns=['Id'])
X_train_pca, X_test_pca, y_train, y_test =
↪  train_test_split(PCA(n_components= 30).fit_transform(
↪  StandardScaler().fit_transform( df.iloc[:,1:].values)), df.iloc[:,0],
↪  test_size=0.2)
```

- Grid search, take random forest as example:

```
X = PCA(n_components = 30).fit_transform( StandardScaler().fit_transform(
↪  df.iloc[:,1:]))
modelRFC = GridSearchCV(RandomForestClassifier(random_state=42),
↪  {'criterion':['gini', 'entropy'], 'max_features': ['auto', 'log2', 0.25,
↪  0.5, 0.75], 'min_samples_split': [2, 4, 6, 8, 10], 'min_samples_leaf':
↪  [1, 3, 5], 'max_depth': [20, 40, 60]}, n_jobs = -1, verbose=10).fit(X,
↪  df.iloc[:,0])
pprint(modelRFC.cv_results_)
pprint(modelRFC.best_params_)
```

- Function to plot the confusion matrix

```
def cm_plot(cm, y_test, title):
    N = list(map(lambda clazz : sum(y_test == clazz), [*range(5)]))
    plt.figure(figsize=(7,5))
    c = plt.pcolormesh([cm[j, :] / N[j] for j in range(5)], vmin=0.0,
    ↪  vmax=1.0)
    plt.title(title)
    plt.colorbar()
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.xticks(0.5 + np.arange(5), np.arange(5))
    plt.yticks(0.5 + np.arange(5), np.arange(5))
    c.update_scalarmappable()
    ax = c.axes
    for p, color, value in zip(c.get_paths(), c.get_facecolors(),
    ↪  c.get_array()):
        x, y = p.vertices[:-2, :].mean(0)
        ax.text(x, y, "%.2f" % value, ha="center", va="center", color=(0.0,
        ↪  0.0, 0.0) if sum(color[:2] > 0.3) >= 2 else (1.0, 1.0, 1.0))
    return plt.show()
```

- Function to plot the ROC curve

```python
def roc_plot(y_test, y_score, name):
    fpr, tpr, roc_auc = dict(), dict(), dict()
    for i in range(5):
        fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    plt.figure(figsize=(7,7))
    plt.plot([0, 1], [0, 1], 'k--')
    [plt.plot(fpr[i], tpr[i], label= 'Class ' + str(i) + ' ROC(area =
    ↪   %0.2f)' % roc_auc[i]) for i in range(5)]
    plt.xlim([-0.02, 1.0])
    plt.ylim([0.0, 1.02])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc="lower right")
    plt.title(name + ' ROC Plot')
    return plt.show()
```

- Function to generate prediction using a trained model

```python
def gen_csv(model, name):
    result = dfPred[['Id']].copy()
    result['Label'] = model.predict(X_pred)
    result.to_csv('result' + name + '.csv', encoding='utf-8', index=False)
```

- Function to calculate Histogram of Oriented Gradients

```python
def calc_hog_features(X):
    return np.array([hog(x.reshape((28, 28)), orientations = 8,
    ↪   pixels_per_cell = (8, 8), cells_per_block=(1, 1)) for x in X])
```

- Fit a model

```python
clf.fit(X_train_pca, y_train)
```

1.3 Kaggle Competition Score

0.90520

Achieved by using PCA n_components = 100, XGBoost with parameters learning_rate = 0.005, n_estimators = 10000, max_depth=15, min_child_weight=1, gamma = 0, colsample_bytree = 0.9, subsample = 0.8, scale_pos_weight=1, reg_alpha = 0. Model was trained on all training data. It took around 5 hours to train on a 40-core CPU.

1.4 Results Analysis

1.4.1 Runtime performance for training and testing

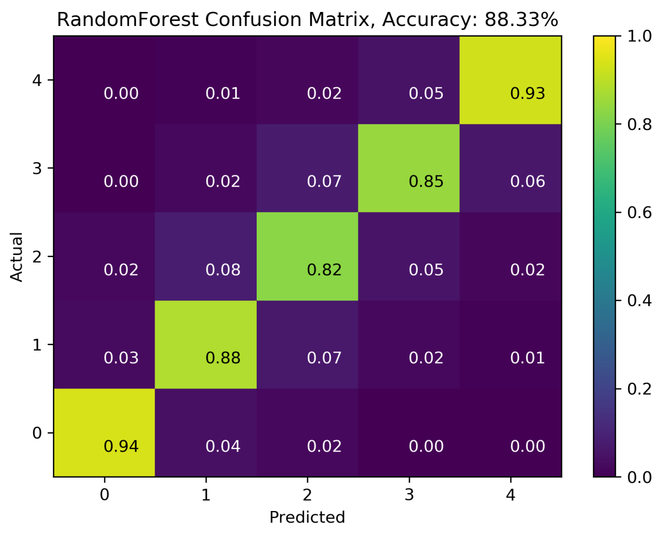
The fitting and predicting time for each method is shown below:

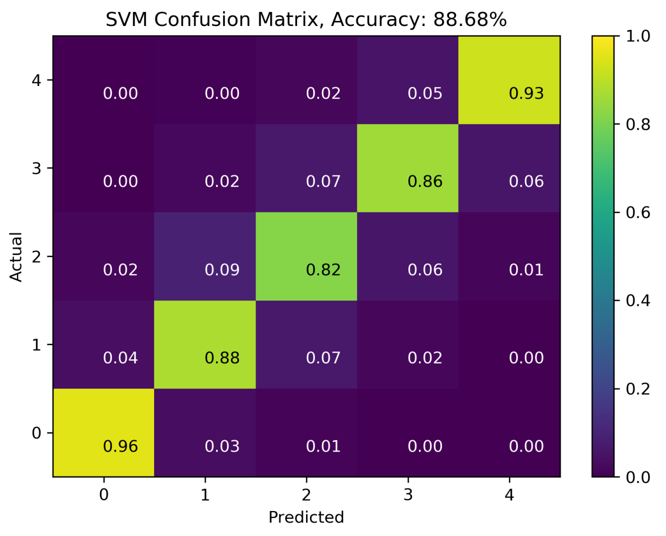
Classifier	**Training(s)**	**Testing(s)**
Random Forest	18.00	0.32
SVM	42.73	8.60
XGBoost	90.60	0.25
Histogram-based Gradient Boosting	42.97	1.06

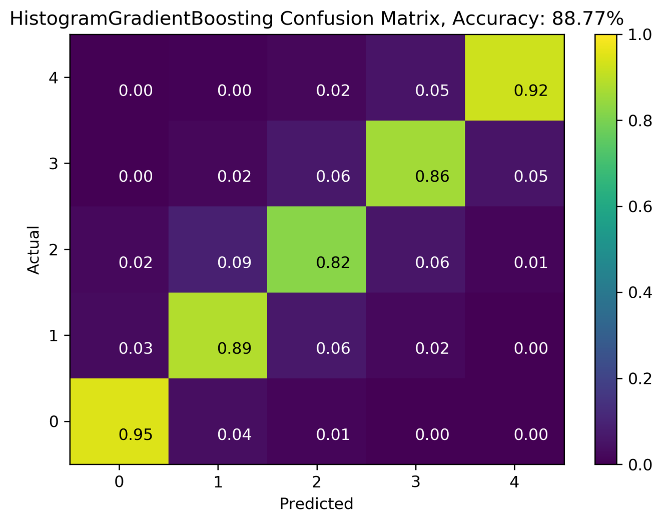
In terms of fitting time, XGBoost is the longest and Random Forest is the least. And SVM takes much longer time to predict new results than other methods. However, the comparison is not that meaningful. First, they achieved different accuracy. I can decrease the estimators in XGBoost and increase its learning rate to make it faster, at the cost of accuracy. Also, because SVM is not an ensemble method, and all other three algorithms can run with multicores. The time will vary a lot if those algorithms are tested on a different computer.

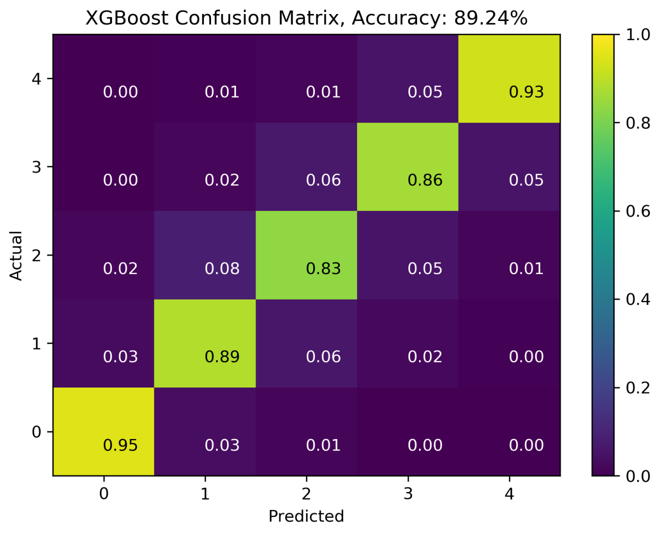
1.4.2 Comparison of the different algorithms and parameters

Using PCA n_components = 30, 80 percent labeled data for training and 20 percent for testing, confusion matrices are plotted as below.









The accuracy given by Kaggle is:

Classifier	**Accuracy**
Random Forest	88.14%
SVM	88.78%
XGBoost	89.38%
Histogram-based Gradient Boosting	89.20%

From the confusion matrices we can see that for all four algorithms, it is easier to classify class 0 and 4 than other 3 classes. The accuracy to classify 0 and 4 is around 93 percent while it is only

82 percent for class 2. We can also see that the similarity between adjacent classes is relatively greater, for most misclassification rate is contributed by the grids next to the minor diagonals on the matrices.

For random forest, the accuracy will not increase any more when the number of estimators increased to a certain amount, no matter what the other parameters are. However, the accuracy of Hist gradient boosting and xgboost will increase gradually when the learning rate is decreased and n_estimator is increased.

The specific parameters I tried for each algorithm will be discussed in the following sub-subsection.

1.4.3 Explanation of your model

Use GridSearchCV(5 folds) to find the best parameters for each algorithm.

Random Forest

- n_estimators: default value(100), we do not have to tune this parameter since we all know a larger value at least will not make the result worse.
- criterion: [gini, entropy], different functions to measure the quality of a split.
- max_features: [auto, log2, 0.25, 0.5, 0.75], number of features to consider when split.
- min_samples_split: [2, 4, 6, 8, 10], min samples required to split a node.
- min_samples_leaf: [1, 3, 5], min samples required to be a leaf node.
- max_depth: [20, 40, 60], max tree depth.

In total 450 combinations. Best parameter: {criterion: entropy, max_depth: 40, max_features: 0.5, min_samples_leaf: 1, min_samples_split: 4}, accuracy: 88.5%.

SVM

- kernel: rbf, image classification is a nonlinear problem so linear kernel was not considered. And the poly kernel was not considered for its low efficiency according to this blog.
- C: [0.01, 0.1, 1, 5, 10, 15, 20, 25, 30, 35], regularization parameter.

In total 10 combinations. Best parameter: {C: 35}, accuracy: 88.56%.

Hist Gradient Boosting

- learning_rate: default value(0.1)
- max_iter: default value(100), lower the learning rate and increase max iteration will almost always improve the result, so I did not tune these two parameters.
- max_leaf_nodes: [35, 37, 39, 41], max leaves for each tree.
- min_samples_leaf: [22, 24, 26, 28], min samples per leaf. Since the training set is large, I tried values greater than the default value(20).

In total 16 combinations. Best parameter: {max_leaf_node: 41, min_samples_leaf: 24}, accuracy: 88.83%

XGBoost

Following the instruction on this blog, I tuned these parameters step by step.

Step 1:

- max_depth: [3, 5, 7, 9, 15], max depth of a tree, typical value 3-10.

- min_child_weight: [1, 3, 5], min sum of weights of all observations required in a child.

Step 2:

- gamma: [0, 0.1, 0.2, 0.3, 0.4], regularisation parameter.

Step3:

- sub_sample: [0.6, 0.7, 0.8, 0.9], subsample ratio, to prevent overfitting.

- colsample_bytree: [0.6, 0.7, 0.8, 0.9], subsample ratio of columns when constructing a tree.

Step 4:

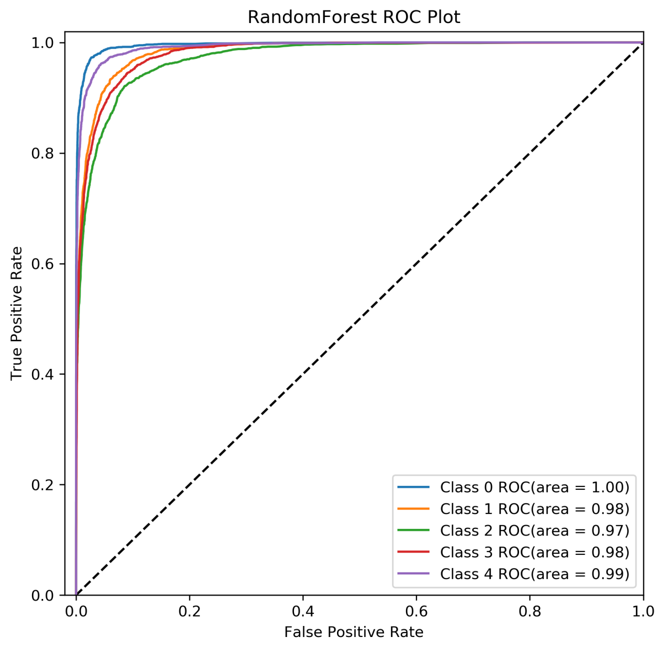
- reg_alpha: [0, 1e-5, 1e-2, 0.1], l1 regularisation parameter, the greater, the more conservative the model will be.

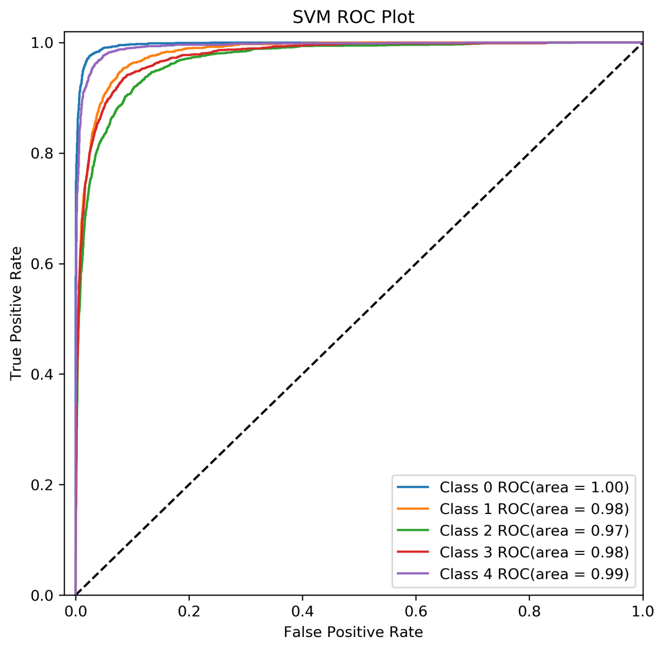
In total 15+5+16+4=40 combinations. Best parameters: {max_depth: 15, min_child_weight: 1, gamma: 0, subsample: 0.8, colsample_bytree: 0.9, reg_alpha: 0}, accuracy: 89.31%.

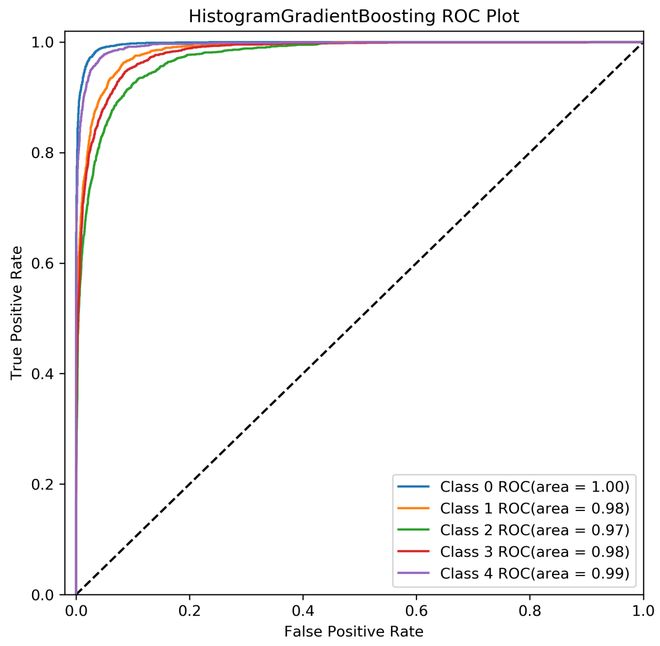
XGBoost has the best performance among all 4 algorithms.

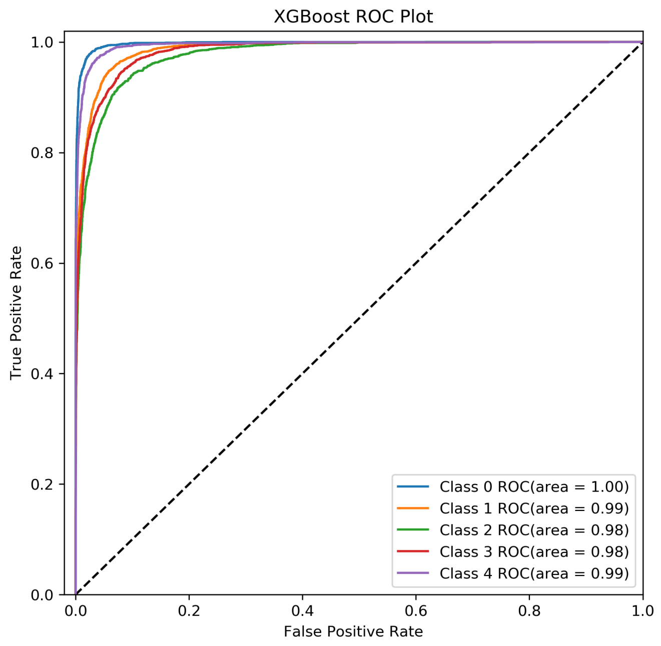
1.4.4 ROC Curve

ROC plots of the four algorithms are shown below, AUC is calculated in the plots' legend.









The ROC curve was plotted using one-vs-all methodology since this is a multi-class model. So there are 5 lines in each plot, each line represents the ROC curve of one class versus the other classes.

We can learn from the plots that the classifiers performed fairly good on this dataset, as the

AUC is close to 1 for all curves. For example, there is 98 percent of chance that the random forest model will be able to distinguish class 1 from other classes.

We also know that the difficulty to separate each class is class $2 > 3 > 1 > 4 > 0$.

1.4.5 Evaluate your code with other metrics on the training data and argue for the benefit of you approach

Previously, we have used confusion matrix, which is another metric, to analyze the result. And here we add precision, recall and f1 score for each model, the scores were calculated for each class.

Precision=$\frac{true\ positive}{true\ positive+false\ positive}$, which indicates the proportion of the data points the model says was relevant actually were relevant.

Recall=$\frac{true\ positive}{true\ positive+false\ negative}$, which indicates the model's ability to find all relavent instances in the dataset.

F1=$2 \cdot \frac{precision+recall}{precision \cdot recall}$, which is the harmonic mean of precision and recall, and it takes both metrics into account.

Model	**Precision(Class 0-4) %**	**Recall(Class 0-4) %**	**F1(Class 0-4) %**
RF	94.63 85.47 82.80 86.14 92.26	93.37 87.74 82.91 84.42 92.75	94.00 86.59 82.86 85.27 92.50
SVM	94.13 85.09 84.17 85.76 93.28	95.03 88.46 81.27 85.14 92.67	94.58 86.74 82.69 85.45 92.98
XGBoost	95.16 87.04 84.50 86.25 92.99	94.49 88.58 83.53 85.95 93.41	94.82 87.81 84.01 86.10 93.20
HGBoost	94.73 86.64 83.68 85.62 92.92	94.49 87.95 82.58 85.70 92.92	94.61 87.29 83.13 85.66 92.92

Using accuracy alone, we do not know the result for each class. And when the classes are unbalanced, accuracy cannot reflect the real performance of the model on certain classes. And F1 score will solve this problem if we know what is our "class of interest", so that we can adjust our algorithm to make the prediction of a certain class more accurate.