Part 0:
Canvas: Project Group 36, Kaggle: PythonIsFun, leaderboard position: 23
Members: Zheng Nie, 90691455, Chaoran Huang, 63607099, Haining Zhou, 95309476
Part 1:

Models	their performance on our own training & validation data (AUC)	the public & private leaderboard data available (2-4 numbers)
KNN	tr: 0.56, va:0.54	Public: 0.54479 Private: 0.56905
Linear classifier	tr: 0.66; va: 0.59	Public: 0.59049 Private: 0.57440
Random forest with selected important features	va: 0.73312...	Public: 0.74909 Private: 0.74812
Blending random forest and gradient boosting weighted ensembles	Individual: RF va: 0.75962… GB va : 0.75287… Blending (0.68RF, 0.32GB): va: 0.76159...	Public: 0.75358 Private: 0.75007

Part 2:
KNN:

Overall, KNN method has a bad performance on high dimension dataset. Since there are 107 features, it is very hard to have an accurate prediction. However, knn should have a good performance when the dataset is crowded and low dimension. Therefore, it is important to try to lower the dimension so that it should be able to have an improved performance.

I tried the whole dataset first but quickly found that it is not a wise choice because for such a big dataset it takes too long to train and the result wouldn't be expected due to the high dimension. Therefore, I set some random ranges. I hope that from randomly chosen ranges, there would be a good result. Finally, I found that when I randomly selected around 10 features, my validation error would converge to 0.455. And I used it (my choice is the first 10 features) as my selected input. Then I ran the K value from [1, 2,5,10,20,50,100,200] to get a best K. My error plot showed to me that when K = 50, it has a better validation error around 0.45.
how was it trained (learning algorithm and software source);

I used the code provided by the Professor for the first time. The result is awful. Therefore, I should use other packages.

The high dimension data causes a very large variance so that it is extremely difficult for knn to achieve an acceptable performance. When I only use 2 or 3 features as input training dataset, it is underfitting. Otherwise, if I choose a lot of features as input training dataset, the variance interrupts the prediction.

Linear Classifier:

There are 2 important things when training a linear classifier: one is feature selection, the other is choosing an appropriate degree. We used *ml.transforms.fpoly()* which limits the value of *(d+1)**m*, meaning that I cannot have a very high dimension. So:

We first set degree = 5 which is fair, and try different number of feature combinations. By observing the validation auc, we finally made sure that selecting 8 features usually gives the best performance. Then we want to make sure the degree (because 5 may not be the best), and we did this mainly by observing the overfitting conditions.

We tried using higher degrees and we observed overfitting. For instance, when selecting features 50~57, we have (Etr = 0.41, Eva = 0.44, AUCtr = 0.62, AUCva = 0.58) for d = 3, (Etr = 0.39, Eva = 0.43, AUCtr = 0.66, AUCva = 0.59) for d = 5, and (Etr = 0.38, Eva = 0.45, AUCtr = 0.67, AUCva = 0.59) for d = 6.

We can see that from d = 3 to d = 5, both training error and validation error decreased and that was what we wanted. But from d = 5 to d = 6, training error still decreased while validation error increased. This indicated the happening of overfitting, so we finally chose d = 5. Unfortunately, the linear classification model did not perform good as well. We think it may still be too simple to predict complex data.

Random forest with selected important features:

We selected 69 out of 107 features to train a Random forest classifier. The idea is to select the features that have a greater impact on the results for further modeling, and ignore the features with less impact.

We used *RandomForestClassifer* in the sklearn module for both feature selection and training since it has better time efficiency for tuning. The selection was based on feature importance evaluation from several different Random forests trained with a full dataset. It was mainly to see how much contribution each feature has made on each tree in the random forest, then take the average, and finally compare the contribution between different features. Then, we trained random forest using the training set and the test set after retaining only the feature columns with high impact rate.

For key hyperparameter settings, the parameter, such as the number of trees and the number of randomly selected features, etc., was tuned accordingly since the dataset had changed after several iterations. We selected the classifier with best_auc to submit on kaggle. It turns out that the classifier has better performance than the homework one.

Part 3:

Blending Random forest and gradient boosting weighted ensembles:

From our general observations, the performance between random forest and gradient boosting classifiers were very close. We wonder if there is a way to combine the success of these two models to better match the test set, and a natural thought is using the weighted average method. Therefore, we eventually tried to make a weighted combination of the training result from those two classifiers to see if this can get us better performance. At the beginning of our experiment, we simply set weight manually (like 0.66:0.37, 0.25:0.75 etc.) and trained them with the full dataset.

It turned out that the blending model could have better performance than any individual performance of the two classifiers if we set the weight properly. After we confirmed this, we wanted to find the most appropriate weight setting to achieve optimal performance. We combined the current best-tuned random forest and gradient descent classifiers using sklearn module. We iterated through 0.00 to 1.00 to find best_weight and record the corresponding local best_auc to generate a few combinations. Comparing the validation results, we selected the best combination and it brought us our best performance on kaggle.

Part 4:

Generally, we think random forest and gradient boosting works particularly well for this data, and KNN and linear models worked poorly. We carefully choose the parameter for the random forest to make sure that our predictions wouldn't be trapped in a certain range because the gradient descent will easily make us only find the local maximum or minimum. After choosing the right parameter we tested, we found that the error rate greatly decreased. We believe that a good parameter allows the model to accept a relatively bad result in order to get better predictions. In addition, the better point of the random forest is that we can trial an error to make a better prediction. Therefore, we choose random forest as the frame to ensemble the model.

For KNN, we used the PCA to lower the dimension of the dataset, However, it is very difficult to find what the appropriate dimension is. We randomly choose the features of the dataset and found that when we apply more than 10 features, the error rate would converge and even if we use all 107 features, the error rate won't have many changes. Therefore, we used the first 10 features.

For the linear classifier, we mainly relied on observing overfitting, which is a common way to select appropriate features and degrees, and that turned out to work well. We improved the validation auc by almost 0.1 by doing this (from 0.50 to 0.59).

Also, for feature selection, since discrete and continuous features coexist in the data set, a natural idea is to normalize the continuous input features to some linear space. Continuous data will become points in [0-1]. Therefore, we can set the threshold between 0-1, which can naturally be handled by binary based algorithms such as decision tree models.