CS 37300 Group Project Final Report

Group Member:

Shupei Wang(wang3107@purdue.edu)

Yuxuan Yang(yang1329@purdue.edu)

Yuxing Chen(chen2689@purdue.edu)

Dataset Description and Pre-processing

We use Higgs Boson dataset from Kaggle. The original dataset contains 250000 samples, each with 30 features and 1 label, 1 event id and 1 weight column. Although there is also another test dataset on the website, it doesn't contain label column so it cannot be used. So we sperate the training set to two parts, one for training and another one for testing. In data pre-processing, we ignored two columns: `eventid` and `weight`. Since the original data set is too large, we select 5000 samples. In the code it's index 120000 to 125000, given the original data should be independent of index. And another 5000 samples from index 125000 to 130000 for testing.

Also, the website states that "it can happen that for some entries some variables are meaningless or cannot be computed; in this case, their value is −999.0, which is outside the normal range of all variables". Regarding the -999 data, we are currently treating it like normal value, we considered getting rid of all these values by deleting all samples that contains a feature that has the value of -999, yet it could though not necessarily lead to bias, since test dataset also contain -999 features.

```python
# Converting original lablels to -1 and 1
s = list(np.where(data=='s'))
data[tuple(s)] = 1
b = list(np.where(data=='b'))
data[tuple(b)] = -1
data1 = data.astype(float)
# Select testing and training samples
data2 = data1[range(120000, 125000)] # testing samples
data1 = data1[range(125000,130000)] # training samples
```

In cleaned dataset, column 0 is `eventid`; 1-30 are features; 31 is `weight` which we will ignore; and 32 is label. After we done pre-processing, `X` contains training samples with features; `y` contains labels of traning samples; `test_X` contains testing samples with features; and `test_y` contains labels of testing samples. Also, Based on the label, we seperate the dataset to positive and negative samples:

```python
X = data1[:,1:31]
y = data1[:,32]
test_X = data2[:, 1:31]
test_y = data2[:,32]

y.astype(int)
y=y.reshape(-1,1)
test_y.astype(int)
test_y.reshape(-1,1)
n,d = X.shape
pos = np.sum(y==1)
neg = np.sum(y==-1)
positive_samples = list(np.where(y==1)[0])
negative_samples = list(np.where(y==-1)[0])
```

Validation and Hyperparamter Tuning

Pre-tuning

At the start of hyperparameter tuning we first use first half of original training samples as training set and the second half as validation set:

```python
train_samples = positive_samples[0:pos/2] + negative_samples[0:neg/2]
validation_samples = positive_samples[pos/2:] + negative_samples[neg/2:]
```

Hyperparameter Tuning

The hyperparameters are `k` (number of nearest neighbors from k-nearest-neighbor) and `F` (number features after reduction in principal component analysis(PCA)). We use the two sets we got above as two folders in two-fold cross validation and we also use nested bootstrapping to tune hyperparameters 'k' and 'F' independently.

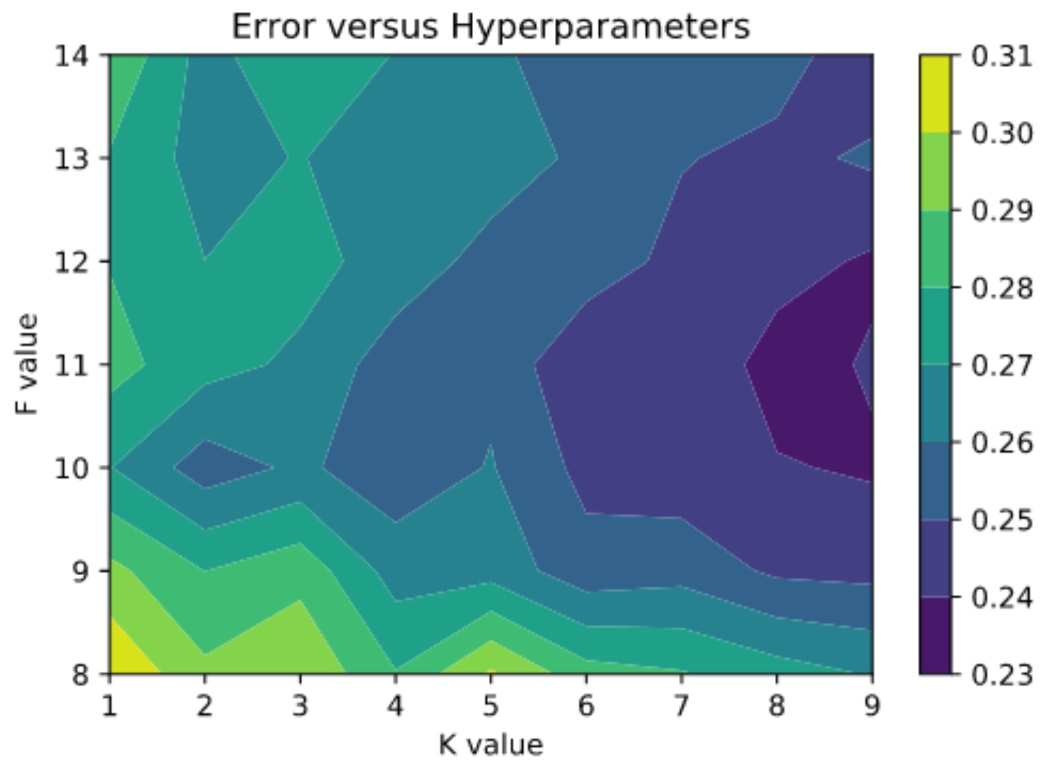
```
for F in range(8,15):
    k_list = list(range(10,30))
    for k in k_list:
        # y_pred = np.zeros(len(X),int)
        mu_fold1, Z_fold1 = pcalearn.run(F, X_fold1)
        X_fold1_small = pcaproj.run(X_fold1, mu_fold1, Z_fold1)
        mu_fold2, Z_fold2 = pcalearn.run(F, X_fold2)
        X_fold2_small = pcaproj.run(X_fold2, mu_fold2, Z_fold2)
        B = 1
        y_pred = np.zeros(len(train_samples) + len(validation_samples), int)
        err = bootstrapping.run(B, X_fold1_small, y[train_samples], k)
        if err < best_err_1:
            best_err_1 = err
            best_k_1 = k
            best_F_1 = F
        temp = err

        err = bootstrapping.run(B, X_fold2_small, y[validation_samples], k)
        if err < best_err_2:
            best_err_2 = err
            best_k_2 = k
            best_F_2 = F
        plot_m[F-8][k-10] = (temp+err)/2
print "Final result is : " + str(best_k_1) + " " + str(best_k_2) + " " +
str(best_err_1) + " " + str(best_err_2) + " " + str(best_F_1) + " " +
str(best_F_2)
```

During hyperparameter tuning, we store the average error we get from two-fold cross validation. To visualize how error change with hyperparameters, we use contour map.

```python
import matplotlib.pyplot as pp
x = range(10,30)
Y = range(8,15)
Z = plot_m
fig,ax = pp.subplots(1,1)
cp = ax.contourf(x, Y, Z)
fig.colorbar(cp)
ax.set_title("Error versus Hyperparameters")
ax.set_xlabel("K value")
ax.set_ylabel("F value")
pp.show()
```

We first use `k` in range of (1,10). However the plot we get look like this:



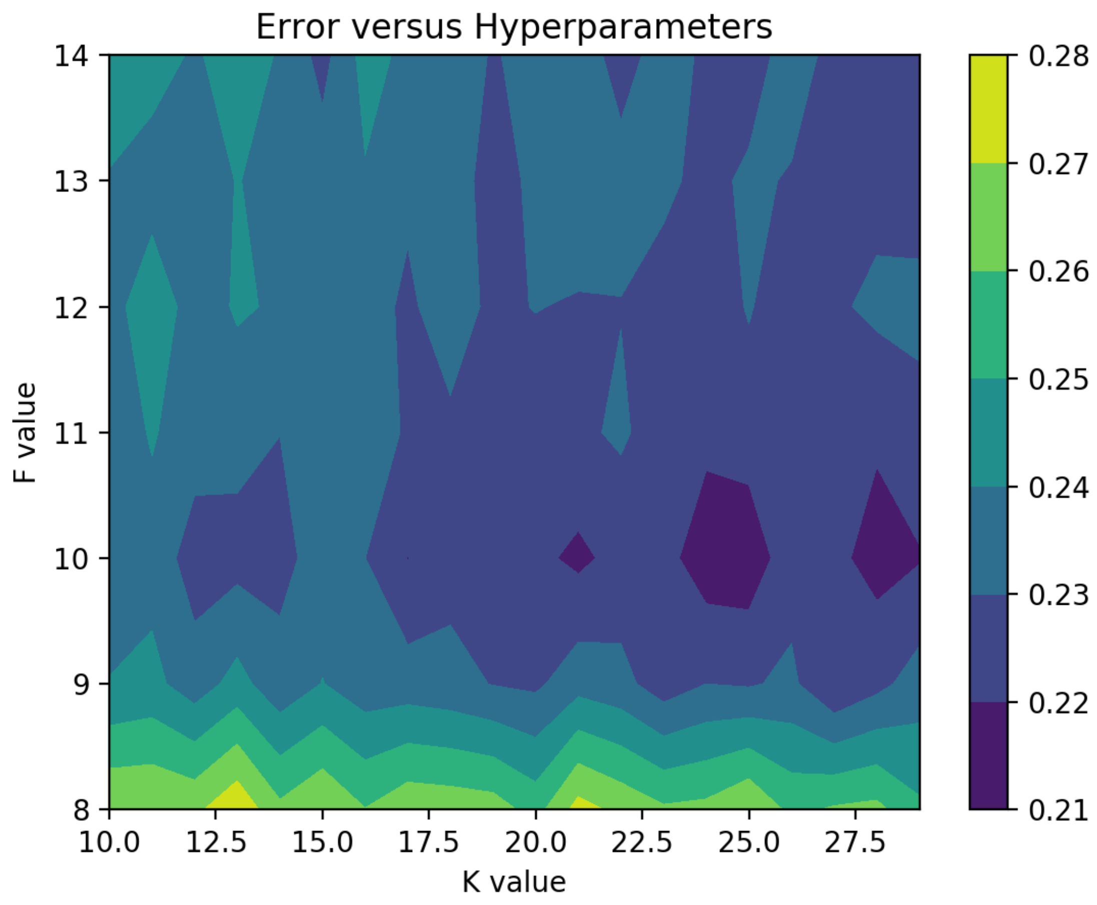
We observe that the error is still decreasing while `k` increasing. So, we decide to increasing `k` value to range(10,30). We will discuss more about the plot in result section.

Results

> Since we generate random numbers in bootstrapping, the graph and result of each run may have slight difference. However, both hyperparameter values will always in the fixed range. And classification error won't change a lot (around 0.22).

Error versus Hyperparameters Plot

After we changed `k` to range(10,30) like we mentioned above, we can get a plot similar to this one. We observe this is more reasonable. We can clearly see the best number of features, `F` is around 10 and best number of nearest neighors , `k`, is around 25.



Classification Error

We also use the best `k` and best `F` we get from hyperparameter tuning for both sets to predict test samples and get classification error:

```
#Find best k and F based on the min error
best_err = np.amin(plot_m)
best_k = np.where(plot_m==best_err)[1][0] + 10
best_F = np.where(plot_m==best_err)[0][0] + 8

mu_fold1, Z_fold1 = pcalearn.run(best_F, X_fold1)
```

```
X_fold1_small = pcaproj.run(X_fold1, mu_fold1, Z_fold1)
mu_fold2, Z_fold2 = pcalearn.run(best_F, X_fold2)
X_fold2_small = pcaproj.run(X_fold2, mu_fold1, Z_fold1)

alg = KNeighborsClassifier(n_neighbors = best_k,algorithm='auto')
alg.fit(X_fold1_small,np.ravel(y[train_samples]))
y_pred[validation_samples] = alg.predict(X_fold2_small)


X_fold1_small = pcaproj.run(X_fold1, mu_fold2, Z_fold2)
X_fold2_small = pcaproj.run(X_fold2, mu_fold2, Z_fold2)

alg = KNeighborsClassifier(n_neighbors = best_k,algorithm='auto')
alg.fit(X_fold2_small,np.ravel(y[validation_samples]))
y_pred[train_samples] = alg.predict(X_fold1_small)

err = np.mean(y != np.array([y_pred]).T)

print "Classification Error:" + str(err)
```

As mentioned above, we have random values in bootstrapping. But the error we get is always around 0.22. Also, one interesting finding is that if we continue increasing `k` range, the best `k` value will conitnue increasing. But the classification error will still be around 0.22. That's why we think range(10,30) is enough for this dataset.

ROC Curve

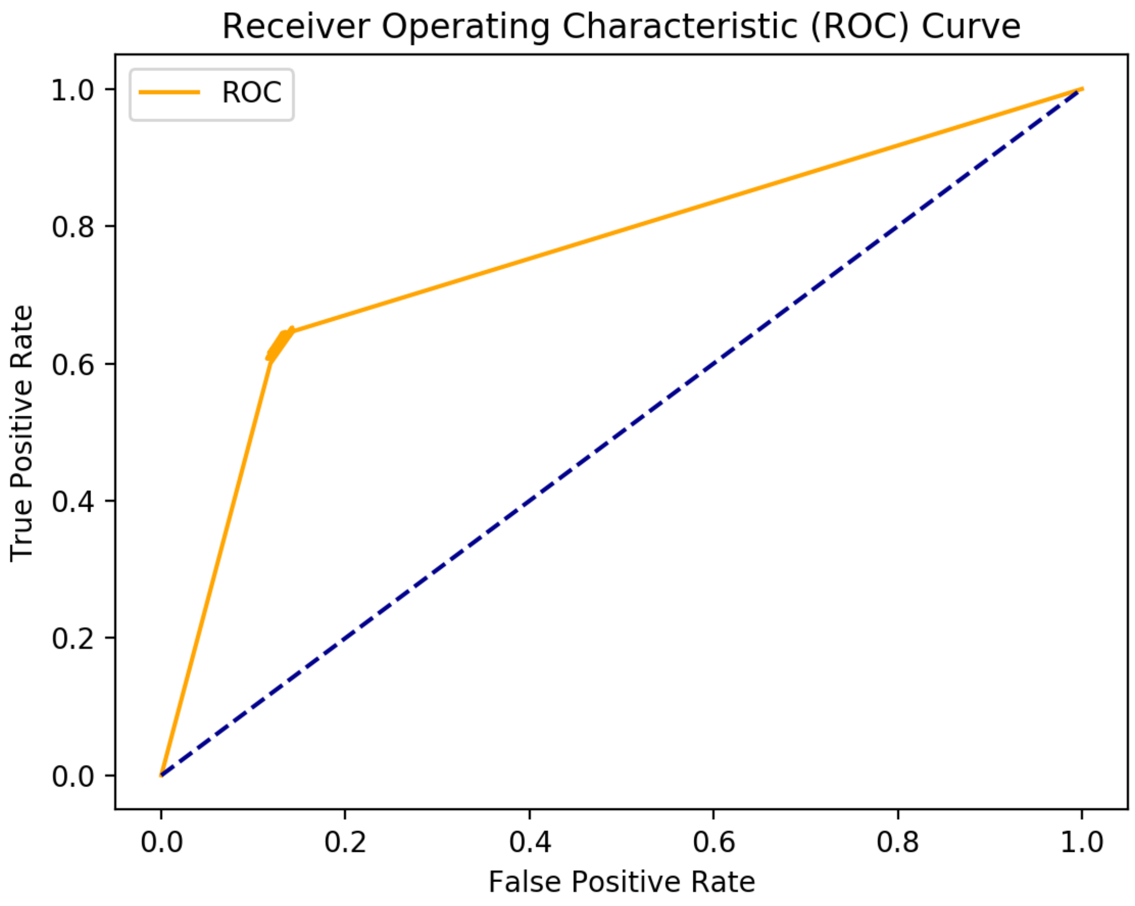
We also implement calculation of TPR and FPR and plot ROC curve:

```
def plot_roc_curve(fpr, tpr):
    plt.plot(fpr, tpr, color='orange', label='ROC')
    plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()

alg = KNeighborsClassifier(n_neighbors = best_k,algorithm='auto')
alg.fit(X,np.ravel(y))
probs = alg.predict_proba(test_X)
probs = probs[:,1]
#print np.array([probs]).T.shape, test_y.shape
fpr = np.zeros(22)
```

```
tpr = np.zeros(22)
P_label = list(np.where(test_y == 1))
P_label = np.asarray(P_label)
P_label = P_label.size
N_label = list(np.where(test_y == -1))
N_label = np.asarray(N_label)
N_label = N_label.size
for i in range(2,30):
    #print i
    alg = KNeighborsClassifier(n_neighbors = i, algorithm='auto')
    alg.fit(X, np.ravel(y))
    probs = alg.predict(test_X)
    TP = 0
    FP = 0
    for j in range(0,len(test_y)):
        if test_y[j] == 1 and probs[j] == 1:
            TP = TP + 1
        if test_y[j] == -1 and probs[j] == 1:
            FP = FP + 1
    fpr[i-9] = FP*1.0/N_label
    tpr[i-9] = TP*1.0/P_label
    #print TP,FP
fpr[0] = 0
fpr[21] = 1
tpr[0] = 0
tpr[21] = 1
print fpr,tpr
plot_roc_curve(fpr,tpr)
```

The curve we get looks like this:



The reason we get a curve like this but not a "smoother" curve may be FPR and TPR won't change a lot with hyperparameter.