Assignment2 - yumin cao, a1754926

## 1 Adaboost Algorithmic Description:

Adaboost was first introduced by Freund and Schapire. It is used for boosting the 'weak learners' to get 'strong learner'.
Supposing given a training set for binary classification

$$T = \{(x_,y_1),(x_2,y_2),...(x_m,y_m)\}$$

Where x belongs to X and y = {-1,+1}, the adaboost algorithm is

Input: T; weak learner
Output: final classifier G(x)

Step1: Initialisation the weight distribution of the training data

$$D_1 = (w_{11},...w_{1i},...w_{1N}),\ w_{1i} = \frac{1}{N},\ i = 1,2,...,N$$

Step2: for m = 1,2,....,M

(1) learning with the dataset that follow the weight distribution Dm, and get the basic classifier, where m is the serial number of related weak learner

$$G_m x : \chi \rightarrow \{-1,+1\}$$

(2) calculate the error rate

$$e_k = P(G_k(x_i) \neq y_i) = \sum_{i=1}^{m} w_{ki} I(G_k(x_i) \neq y_i)$$

(3) calculate the coefficient of Gm(x)

$$\alpha_k = \frac{1}{2} log \frac{1-e_k}{e_k}$$

(4) update the weight distribution, Z is the regularisation factor

$$Z_k = \sum_{i=1}^{m} w_{ki} exp(-\alpha_k y_i G_k(x_i))$$
$$D(k) = (w_{k1}, w_{k2}, ...w_{km})$$
$$w_{k+1,i} = \frac{w_{ki}}{Z_K} exp(-\alpha_k y_i G_k(x_i))$$

Step3: Get the final classifier

$$f(x) = sign(\sum_{k=1}^{K} \alpha_k G_k(x))$$

For step2(2), $\sum_{i=1}^{N} w_{mi} = 1$ . For step2(3), we can know a>=0 when e<= 1/2, and a increases when e decrease. For step2(4), when G(xi) = yi, it means correct classified and yiG(xi) = 1, when G(xi) != yi, it means wrong classified and yiG(xi) = -1.

**2. Understand of Adaboost**

Why adaboost can reduce error?
Adaboost classifier has upper boundary

$$\frac{1}{N}\sum_{i=1}^{N} I(G(x_i) \neq y_i) \leq \frac{1}{N}\sum_{i} exp(-y_i f(x_i)) = \prod_{m} Z_m$$

For binary classification problem,

$$\prod_{m=1}^{M} Z_m = \prod_{m=1}^{M} \sqrt{(1-4\gamma_m^2)} \leq exp(-2\sum_{m=1}^{M}\gamma_m^2)$$

Where r = 1/2 - e

Adaboost algorithm is a special method of forward statewide algorithm(FSA), https://en.wikipedia.org/wiki/Forward_algorithm. The idea of FSA is that if a model can learn from the front to end, and every step only study one basis function and its coefficient. Then we can optimise the loss function. So in adaboost algorithm:

$$f_k(x) = f_{k-1}(x) + \alpha_k G_k(x)$$

$$\underbrace{arg\ min}_{\alpha,G} \sum_{i=1}^{m} exp(-y_i f_k(x))$$

$$(\alpha_k, G_k(x)) = \underbrace{arg\ min}_{\alpha,G} \sum_{i=1}^{m} exp[(-y_i)(f_{k-1}(x) + \alpha G(x))]$$

And finally we can get

$$G_k(x) = \underbrace{arg\ min}_{G} \sum_{i=1}^{m} w'_{ki} I(y_i \neq G(x_i))$$

$$\sum_{i=1}^{m} w'_{ki} exp(-y_i \alpha G(x_i)) = \sum_{y_i = G_k(x_i)} w'_{ki} e^{-\alpha} + \sum_{y_i \neq G_k(x_i)} w'_{ki} e^{\alpha}$$

(1)

$$= (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{m} w'_{ki} I(y_i \neq G_k(x_i)) + e^{-\alpha} \sum_{i=1}^{m} w'_{ki}$$

(2)

After derivation of a, we can get the a that make the loss function minimal, where

$$\alpha_k = \frac{1}{2} log \frac{1-e_k}{e_k}$$

**3. Analysis , experiments**
number of measures/settings that you can report against (training time, prediction on testing set, test time, number of boosting, depth of weak learners – your implementation only has to provide for Stumps

In our class discussion board, the teacher said if we used third packages in tree stump, we should implement it. I didn't used any third packages in tree stump, and I do it with python.

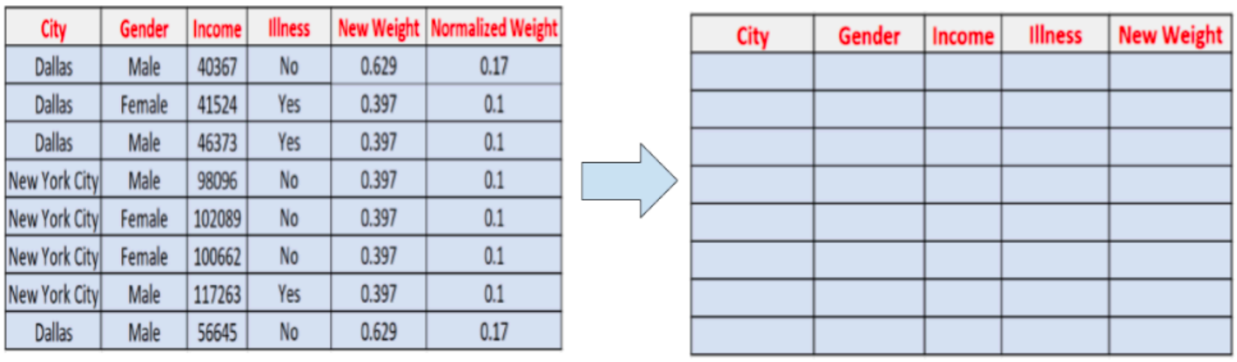
preprocessing data: the label values are 'M' and 'B', and I changed M to 1 and B to -1.
My tree stump is at Adaboost.Gini_CART_DT(), and I used the simplest decision tree model. For each sorted x column, **30 thresholds** are got with same interval(10).
For each threshold, there are two possibilities: G(x)-> 1  x<threshold, G(x)-> -1 x>threshold
or G(x)-> -1 x<threshold, G(x)-> 1 x<threshold
I choose one of the two possibilities with lowest error. The error is the sum of weights of misclassified samples.
I also reflect this in my Adaboost.Gx1(threshold, x) and Adaboost.Gx2(threshold, x) functions for final classifier.

Because the data has continuous values, actually in my first attempt, I used the strategy:

$$threshold = \frac{a_i + a_{i+1}}{2}$$

For 300 training data, I got 299 threshold. But my training errors are very high and **stuck in 0.5.**
Then I realised that we may need to avoid overfitting, such as in decision tree, we need pruning to remove some thresholds. So I chose 30 thresholds to reflect the detail.

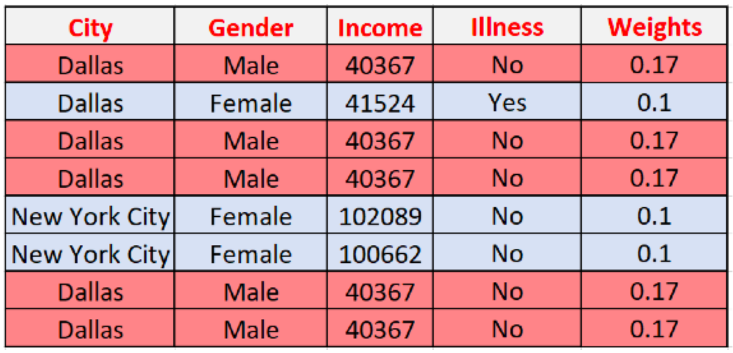
My Adaboost.get_final_classifier() function is for finding the final G(x).
I have seen plenty of web sources calculating the weight distribution, but do not update the data table.
If an instance's weight increasing from 0.1 to 0.2, then it should occur twice time as before. So each time I update the weight, I also update the training set, with higher weight occurs more times and lower weight occurs less times, just like the following( picture from https://medium.com/analytics-vidhya/add-power-to-your-model-with-adaboost-algorithm-ff3951c8de0):



| City | Gender | Income | Illness | New Weight | Normalized Weight |
|---|---|---|---|---|---|
| Dallas | Male | 40367 | No | 0.629 | 0.17 |
| Dallas | Female | 41524 | Yes | 0.397 | 0.1 |
| Dallas | Male | 46373 | Yes | 0.397 | 0.1 |
| New York City | Male | 98096 | No | 0.397 | 0.1 |
| New York City | Female | 102089 | No | 0.397 | 0.1 |
| New York City | Female | 100662 | No | 0.397 | 0.1 |
| New York City | Male | 117263 | Yes | 0.397 | 0.1 |
| Dallas | Male | 56645 | No | 0.629 | 0.17 |

| City | Gender | Income | Illness | New Weight |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

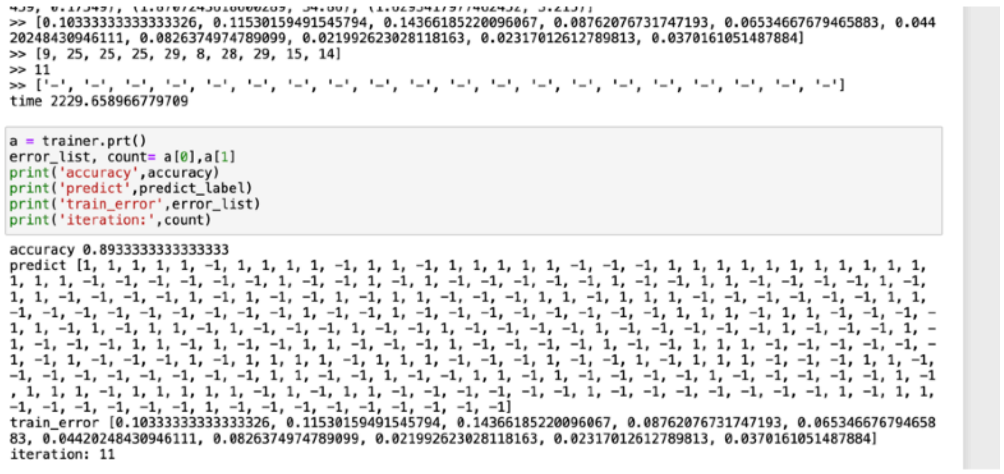
After update:



| City | Gender | Income | Illness | Weights |
|---|---|---|---|---|
| Dallas | Male | 40367 | No | 0.17 |
| Dallas | Female | 41524 | Yes | 0.1 |
| Dallas | Male | 40367 | No | 0.17 |
| Dallas | Male | 40367 | No | 0.17 |
| New York City | Female | 102089 | No | 0.1 |
| New York City | Female | 100662 | No | 0.1 |
| Dallas | Male | 40367 | No | 0.17 |
| Dallas | Male | 40367 | No | 0.17 |

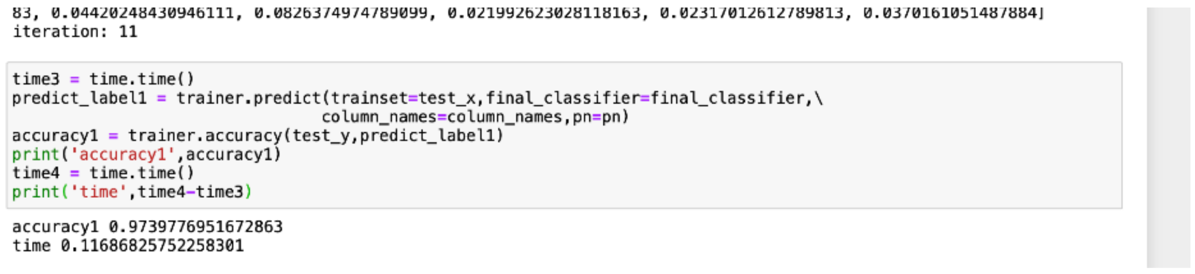
In adaboost.get_final_classifier( ), I set error > 0.005, because 2/300 = 0.006, which means the model has two wrong classified instances. I think making no error for a threshold is maybe too strict, and thus may not perform good on test data, so I allow my model has one wrong value.

After 11 iterations, my error satisfies the condition(<0.005), and for training data, my accuracy is 0.893, using 2229.65 seconds.



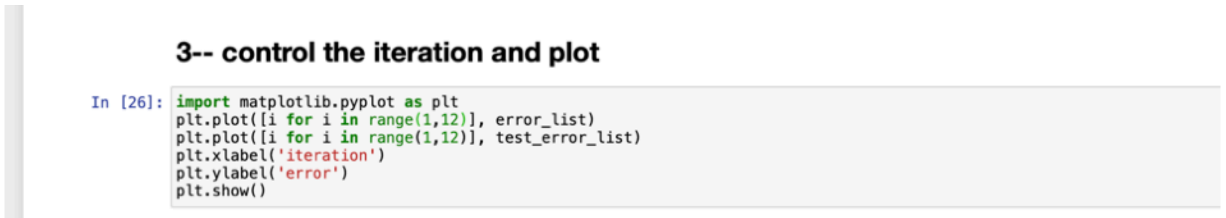
```
>> [0.10333333333333326, 0.11530159491545794, 0.14366185220096067, 0.08762076731747193, 0.06534667679465883, 0.044
20248430946111, 0.0826374974789099, 0.021992623028118163, 0.02317012612789813, 0.0370161051487884]
>> [9, 25, 25, 25, 29, 8, 28, 29, 15, 14]
>> 11
>> ['-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-']
time 2229.658966779709

In [6]: a = trainer.prt()
        error_list, count= a[0],a[1]
        print('accuracy',accuracy)
        print('predict',predict_label)
        print('train_error',error_list)
        print('iteration:',count)

accuracy 0.8933333333333333
predict [1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, -1, 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, -1, -1, -1, -1, -1, -1, 1, -1, -1, 1, -1, 1, -1, -1, -1, -1, -1, 1, -1, -1, 1, 1, -1, -1, -1, -1, 1, -1,
1, 1, -1, -1, -1, -1, 1, -1, 1, -1, -1, 1, -1, 1, 1, -1, -1, -1, 1, 1, -1, 1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, 1, -1, -1, 1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, -1, 1, 1, -1, -1, -1, -
1, 1, -1, 1, -1, 1, 1, -1, 1, -1, -1, -1, 1, -1, -1, 1, -1, -1, -1, -1, 1, -1, -1, -1, -1, -1, 1, -1, -1, -1, 1, -
1, -1, -1, -1, 1, 1, -1, 1, -1, -1, 1, 1, -1, -1, -1, 1, -1, -1, -1, -1, 1, -1, -1, 1, 1, 1, -1, -1, -1, -1, -1, -
1, -1, 1, -1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, -1, -1, 1, -1, -1, 1, -1, 1, 1, 1, -1, -1, -1, 1, 1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, 1, 1, -1, -1, 1, -1, -1, 1, 1, -1, 1, -1, -1, -1, -1, 1, -1, -1, -1, -1, -1, 1, -1
, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, -1, 1, 1, -1, -1, -1, -1, -1, -1, 1, -1, -1, -1, -1, -1, -1, -1, 1, -1, 1, 1,
-1, -1, -1, -1, -1, -1, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
train_error [0.10333333333333326, 0.11530159491545794, 0.14366185220096067, 0.08762076731747193, 0.065346676794658
83, 0.04420248430946111, 0.0826374974789099, 0.021992623028118163, 0.02317012612789813, 0.0370161051487884]
iteration: 11
```

For testing data, my accuracy is 0.974, and test time is 0.1 second.



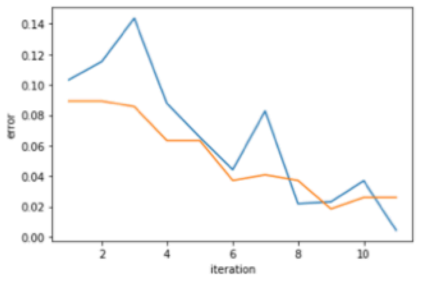
```
83, 0.04420248430946111, 0.0826374974789099, 0.021992623028118163, 0.02317012612789813, 0.0370161051487884]
iteration: 11

In [7]: time3 = time.time()
        predict_label1 = trainer.predict(trainset=test_x,final_classifier=final_classifier,\
                                         column_names=column_names,pn=pn)
        accuracy1 = trainer.accuracy(test_y,predict_label1)
        print('accuracy1',accuracy1)
        time4 = time.time()
        print('time',time4-time3)

accuracy1 0.9739776951672863
time 0.11686825752258301
```

Then we can get the curve:

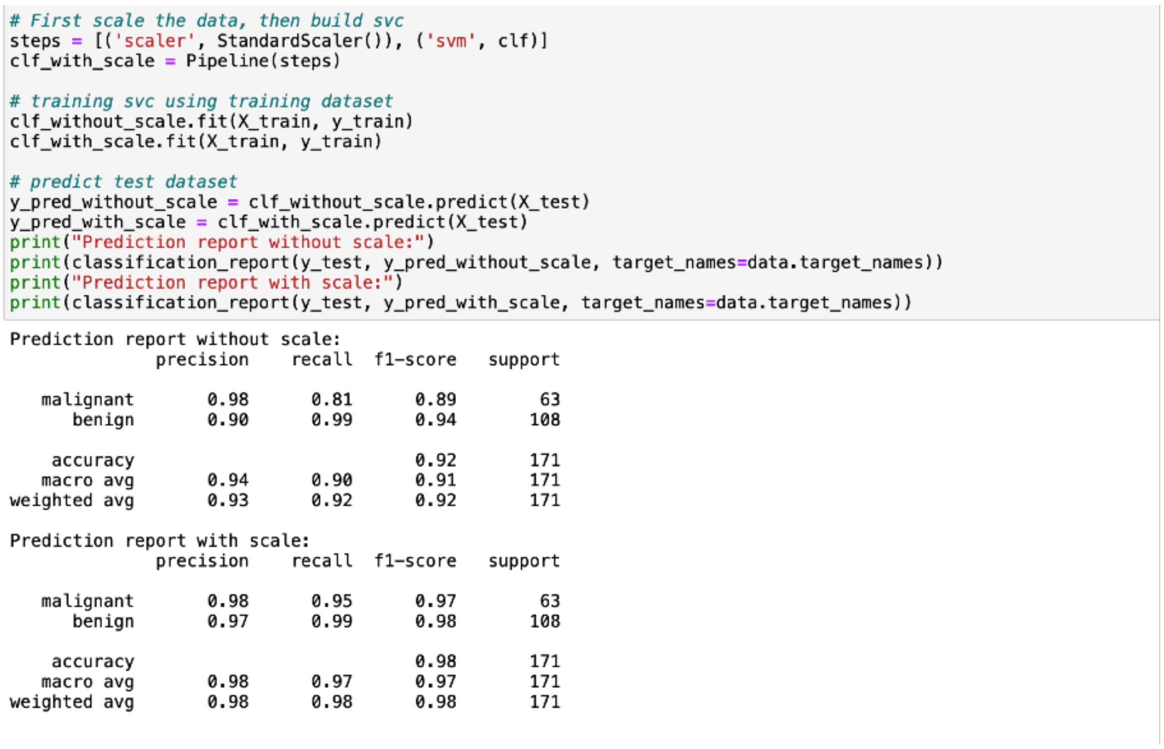


### 3-- control the iteration and plot

```
In [26]: import matplotlib.pyplot as plt
         plt.plot([i for i in range(1,12)], error_list)
         plt.plot([i for i in range(1,12)], test_error_list)
         plt.xlabel('iteration')
         plt.ylabel('error')
         plt.show()
```



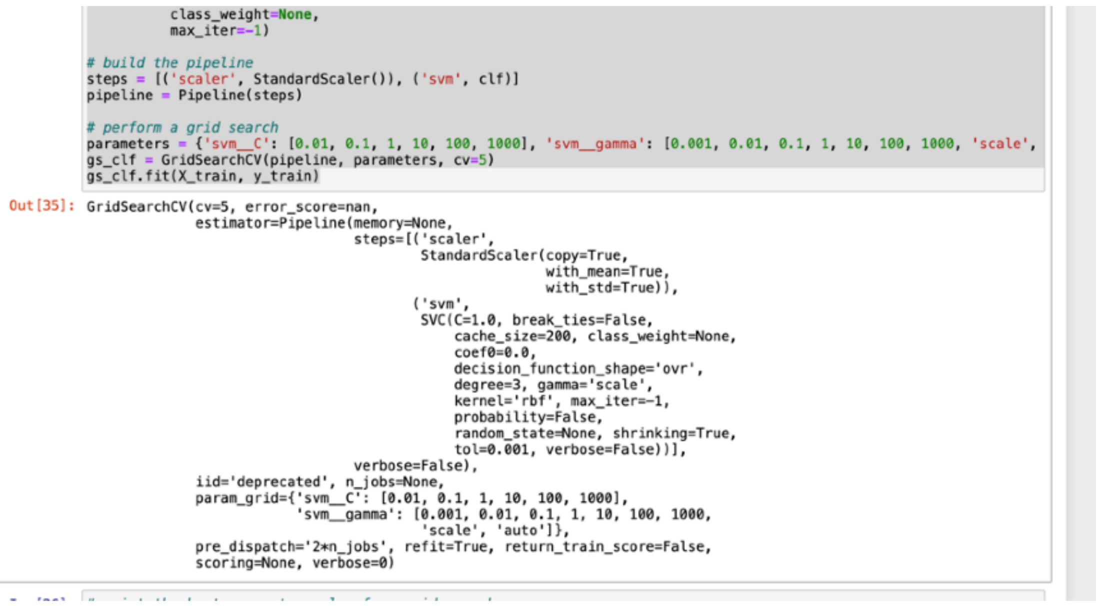
4. SVM
In the first part of SVM, I compared the method with scale and without scale, and find that the SVM with scaling has higher accuracy.
For the SVM, the data needs to be scaled for SVM classification because we can see that the range of different features varies significantly, if we do not scale the data, the data points will live in a very small portion of the feature space, and make it harder for SVM to build a classification boundary. In the following experiment, we can see that after scaling the data, the overall accuracy boosted from 0.92 to 0.98.



```
# First scale the data, then build svc
steps = [('scaler', StandardScaler()), ('svm', clf)]
clf_with_scale = Pipeline(steps)

# training svc using training dataset
clf_without_scale.fit(X_train, y_train)
clf_with_scale.fit(X_train, y_train)

# predict test dataset
y_pred_without_scale = clf_without_scale.predict(X_test)
y_pred_with_scale = clf_with_scale.predict(X_test)
print("Prediction report without scale:")
print(classification_report(y_test, y_pred_without_scale, target_names=data.target_names))
print("Prediction report with scale:")
print(classification_report(y_test, y_pred_with_scale, target_names=data.target_names))
```

```
Prediction report without scale:
              precision    recall  f1-score   support

   malignant       0.98      0.81      0.89        63
      benign       0.90      0.99      0.94       108

    accuracy                           0.92       171
   macro avg       0.94      0.90      0.91       171
weighted avg       0.93      0.92      0.92       171

Prediction report with scale:
              precision    recall  f1-score   support

   malignant       0.98      0.95      0.97        63
      benign       0.97      0.99      0.98       108

    accuracy                           0.98       171
   macro avg       0.98      0.97      0.97       171
weighted avg       0.98      0.98      0.98       171
```

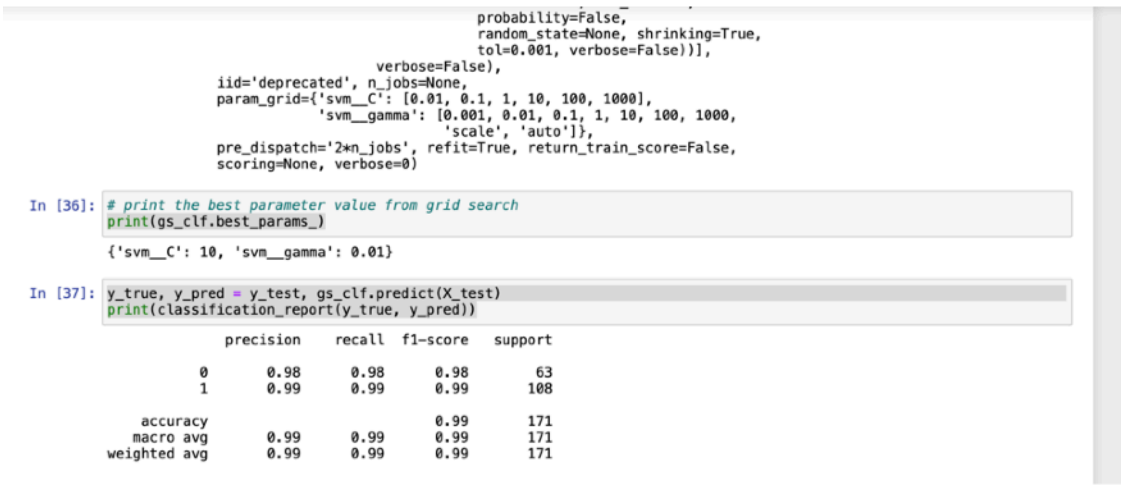
In the second part of SVM, I used grid search to find the best parameters:



```
           class_weight=None,
           max_iter=-1)

# build the pipeline
steps = [('scaler', StandardScaler()), ('svm', clf)]
pipeline = Pipeline(steps)

# perform a grid search
parameters = {'svm__C': [0.01, 0.1, 1, 10, 100, 1000], 'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100, 1000, 'scale',
gs_clf = GridSearchCV(pipeline, parameters, cv=5)
gs_clf.fit(X_train, y_train)
```

```
Out[35]: GridSearchCV(cv=5, error_score=nan,
             estimator=Pipeline(memory=None,
                                steps=[('scaler',
                                        StandardScaler(copy=True,
                                                       with_mean=True,
                                                       with_std=True)),
                                       ('svm',
                                        SVC(C=1.0, break_ties=False,
                                            cache_size=200, class_weight=None,
                                            coef0=0.0,
                                            decision_function_shape='ovr',
                                            degree=3, gamma='scale',
                                            kernel='rbf', max_iter=-1,
                                            probability=False,
                                            random_state=None, shrinking=True,
                                            tol=0.001, verbose=False))],
                                verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'svm__C': [0.01, 0.1, 1, 10, 100, 1000],
                         'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100, 1000,
                                        'scale', 'auto']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

The best parameters of SVM and its accuracy are:



```
                                              probability=False,
                                              random_state=None, shrinking=True,
                                              tol=0.001, verbose=False))],
                                    verbose=False),
                     iid='deprecated', n_jobs=None,
                     param_grid={'svm__C': [0.01, 0.1, 1, 10, 100, 1000],
                                 'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100, 1000,
                                                'scale', 'auto']},
                     pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                     scoring=None, verbose=0)
```

```
In [36]: # print the best parameter value from grid search
         print(gs_clf.best_params_)

         {'svm__C': 10, 'svm__gamma': 0.01}

In [37]: y_true, y_pred = y_test, gs_clf.predict(X_test)
         print(classification_report(y_true, y_pred))

                       precision    recall  f1-score   support

                    0       0.98      0.98      0.98        63
                    1       0.99      0.99      0.99       108

             accuracy                           0.99       171
            macro avg       0.99      0.99      0.99       171
         weighted avg       0.99      0.99      0.99       171
```

As we all know, the SVM with kernel is a strong learner and we have prove that in Assignment1, but the results of SVM and Adaboost are similar(accuracy 0.98, 0.97). This is what we have said in the former. The Adaboost algorithm combines the weak learners (very simple tree stumps) to generate a strong learner,  which has the ability to predict very precisely.