MIDDLE EAST TECHNICAL UNIVERSITY
NORTHERN CYPRUS CAMPUS

CNG 562
MACHINE LEARNING

Midterm 1

Report

Kaan Taha Köken 2152064

April 21, 2020

Contents

List of Figures

1 Findings

In this experiment, first, I tried with different validation methods using base models. Then, I built my both model according to raw data. I continued with cleaning my Breast Cancer data. I made experience with both clean and raw data using both baby sited models. Then, I chose SVM as winner because it was giving more consistent results. Lastly, I applied both boosting and bagging methods. Boosting, slightly boosted the results, but bagging made it worse.

2 Introduction

In this take home exam assignment, our aim is to use methodologies learned earlier such as prepossessing and validation techniques with two classification methods, which are Support Vector Machine (SVM) and Random Forest. We are also aiming to apply bagging or boosting algorithm to the best.

2.1 Dataset

We are using Breast Cancer dataset for the take home exam assignment. It contains 2 classes, 569 instances with 30 features. We can understand by looking its shape. This is the shape of our data (569, 30).

In order to understand and get better perspective from our data. We project our data graph, and see how it looks.

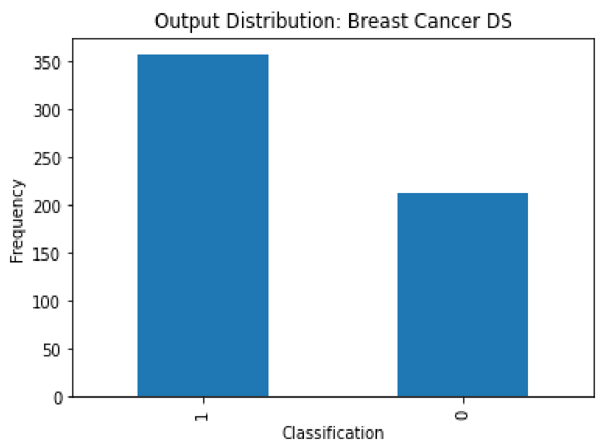

Figure 1: Data visualization

2.2 Data Cleaning

I conduct my experiments in two parts, one with raw data and one with cleaned data. In order to clean my data, I came up with some methods. First, I tried to identify the noisy data inside the data set.

```
def identify_noise(df):

        noise = df[df.isnull().any(axis=1)].count()
        total_noise = noise.sum()
        print("{0} null values were found.".format(str(total_noise)))
        if(total_noise > 0):
                print(noise)
        print("\n\nShowing all data types:\n\n")
        print(df.dtypes)
```

Figure 2: Identify Noise Method

As a result, as we can see result below, there is no noisy data, and we can also see the data types of features.

```
0 null values were found.

Showing all data types:

0     float64
1     float64
2     float64
3     float64
4     float64
5     float64
6     float64
7     float64
8     float64
9     float64
10    float64
11    float64
12    float64
13    float64
14    float64
15    float64
16    float64
17    float64
18    float64
19    float64
20    float64
21    float64
22    float64
23    float64
24    float64
25    float64
26    float64
27    float64
28    float64
29    float64
dtype: object
```

Figure 3: Identify Noise Method

To see correlation in the data, I created a method, and visualize the data as a heat map. As we can see in below, the feature pairs that have a white color are highly correlated. Therefore, correlation will cause a problem for our models. For example, feature pairs (0, 2), (0, 3), and (2, 3) are all highly correlated. If we do not removed them, models will be highly correlated.

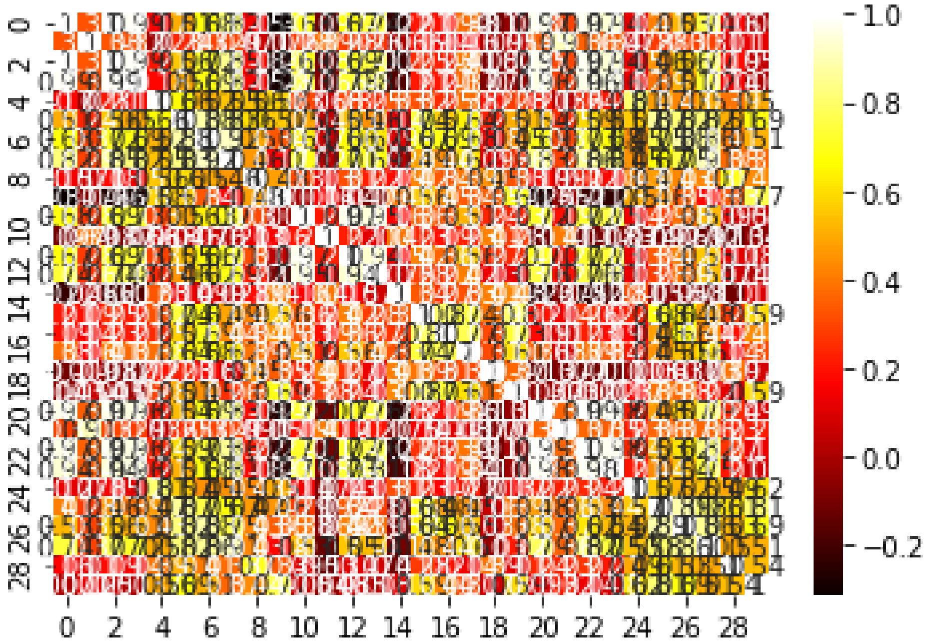

Figure 4: Heat map

```
def heat_map(df):
        fig, ax = plt.subplots()
        corr = df.corr()
        sns.heatmap(corr, annot=True, cmap='hot')
        plt.show()
```

```
def filter_features(data, bad_indices):
        # eliminate above column indices from the data and return new set
        filtered_data = np.delete(data, bad_indices, axis=1)

        return filtered_data
```

Figure 5: Heat Map and Filter Code

After done some cleaning using code above, again, we need to look graphs if there is a distinct difference between labels. For example, feature 7, looks like have distinct difference, so in the future, it will not be useful. Therefore, we need to remove it as well.

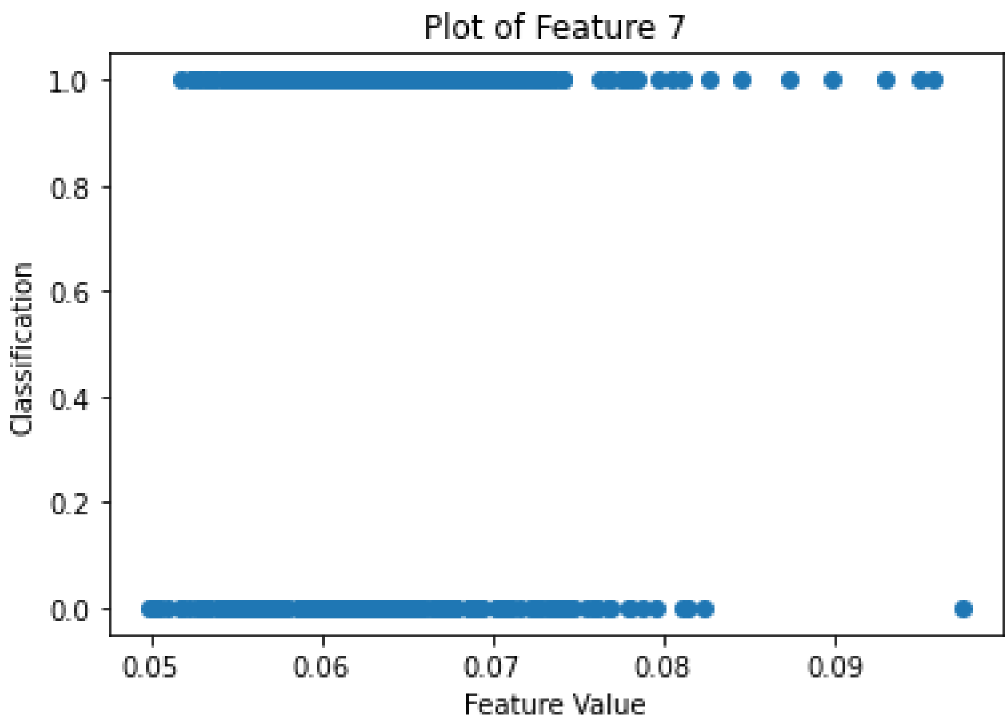

Figure 6: Feature 7

After the last cleaning, we have remained 14 features. Using the pairplot functionality, I display the data remain.

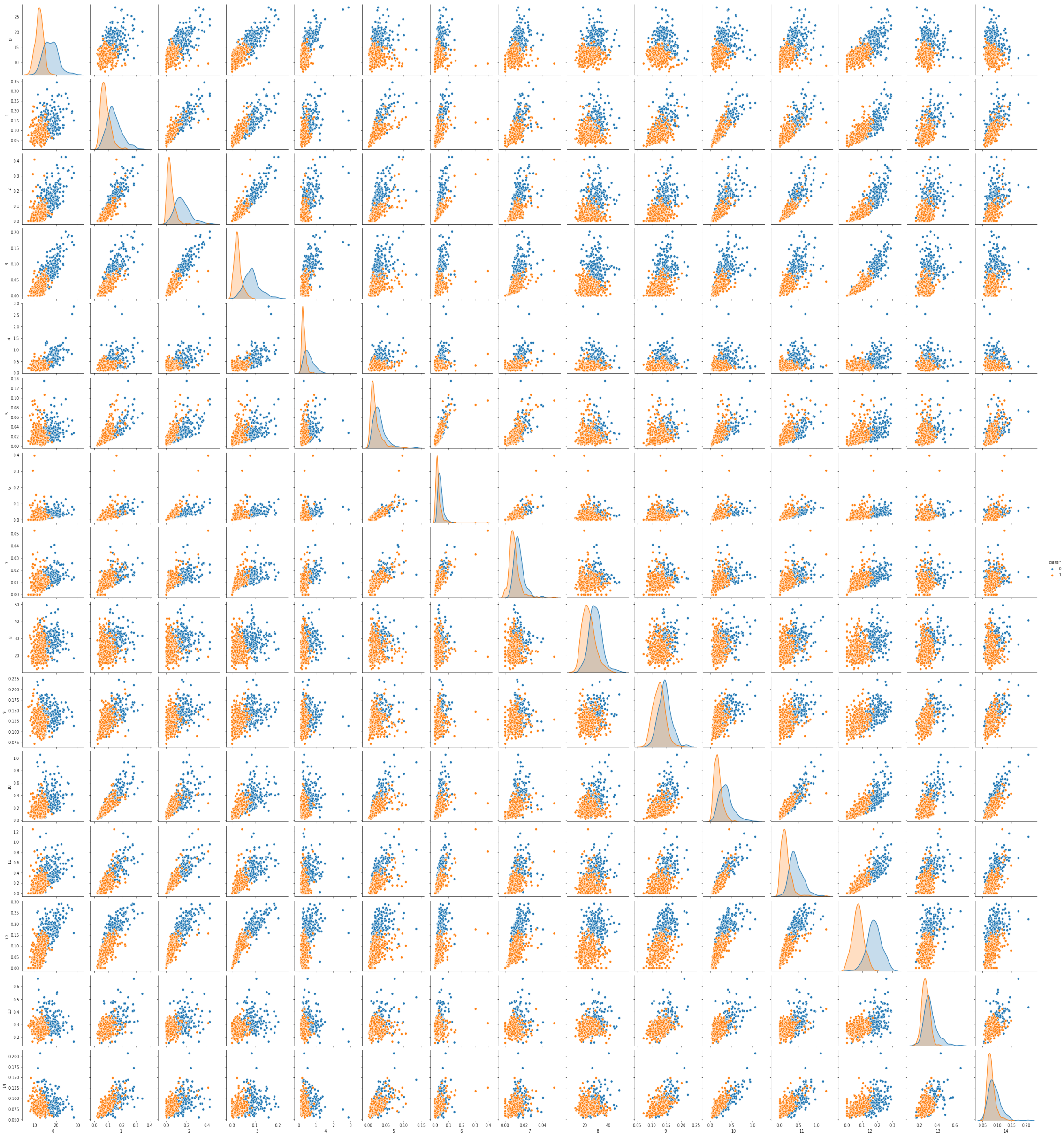

Figure 7: Pair plot

2.3 Validation

2.3.1 Random 1-Hold Out

Random 1-hold out is basically splitting up the dataset into a ‘train’ and ‘test’ set randomly. The training set is what the model is trained on, and the test is used to see performance of the model on unseen data.

2.3.2 Stratified 1-Hold Out

Stratified 1-hold out is splitting up the dataset into a 'train' and 'test' set so that each split has same percentage of samples of each targets as the complete set.

2.3.3 Random k-Fold

Random k-fold is a cross-validation technique which splits up the dataset into 'k' groups. One of the groups is used as the test set and the rest are used as the training set. The model is trained on the training set and scored on the test set. The process is repeated until each unique group as been used as the test set.

3 Random Forest

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes or mean prediction of the individual trees. We will change it parameters such as weight, depth, and will try to find the best model.

3.1 Split Metrics

3.1.1 Gini

A Gini score gives an idea of how good a split is by how mixed the classes are in the two groups created by the split. A perfect separation results in a Gini score of 0, whereas the worst case split that results in 50/50 classes.

$$Gini = 1 - \sum_j p_j^2$$

Figure 8: Gini Calculation

3.1.2 Entropy

Entropy can be roughly thought of as how much variance the data has. It gives the homogeneity of a sample. If the sample is completely homogeneous the entropy is zero and if the sample is equally divided then it has entropy of one.

$$Entropy = -\sum_j p_j \log_2 p_j$$

Figure 9: Entropy Calculation

3.2 Experiments

First, we divided our data 70 percent as train data and 30 percent as test data. Then, we used different validation techniques on our train and test data to see which one is given the best result among them. For this purpose, we created a method called random_forest.

```
def random_forest(X_train, Y_train, forest_size, max_depth, criterion, min_samples_split, class_we
ight):
    rf = RandomForestClassifier(n_estimators=forest_size, oob_score=True, n_jobs=-1, max_depth=max
_depth, criterion=criterion, min_samples_split = min_samples_split, class_weight=class_weight, ran
dom_state=0)

            #5-Fold
    cv_result_rf_5 = cross_val_score(rf, X_train, Y_train, cv=5, scoring='accuracy')

    #10-Fold
    cv_result_rf_10 = cross_val_score(rf, X_train, Y_train, cv=10, scoring='accuracy')

    #Random One Holdout
    x_train, x_test, y_train, y_test_random = randomOneHoldout(X_train, Y_train)
    rf.fit(x_train, y_train)
    y_pred_rf_random = rf.predict(x_test)

    #Stratified One Holdout
    x_train, x_test, y_train, y_test_stratified = stratifiedOneHoldout(X_train, Y_train)
    rf.fit(x_train, y_train)
    y_pred_rf_stratified = rf.predict(x_test)

    print("Stratified One Hold Out Fold")
    print("Random Forest  Accuracy: ", 1 - metrics.mean_squared_error(y_test_stratified, y_pred_rf
_stratified))

    print("5 Fold")
    print("Random Forest Accuracy: ", cv_result_rf_5.mean())

    print("10 Fold")
    print("Random Forest  Accuracy: ", cv_result_rf_10.mean())

    print("Random One Hold Out")
    print("Random Forest  Accuracy: ", 1 - metrics.mean_squared_error(y_test_random, y_pred_rf_ran
dom))
```

Figure 10: Random Forest Code

Without manipulating the data, with base model of random forest, we got this accuracy. After the result, I decided to with ***Stratified One Hold Out***

```
Stratified One Hold Out Fold
Random Forest  Accuracy:  0.975
5 Fold
Random Forest Accuracy:  0.959873417721519
10 Fold
Random Forest  Accuracy:  0.9523717948717948
Random One Hold Out
Random Forest  Accuracy:  0.975
```

Figure 11: Random Forest Raw Result

Then, I started to change its parameter individually. First, I started play with **depth** parameter. To do that, I created a method called **tuningDepth**.

```
def tuningDepth(x_train, x_test, y_train, y_test_stratified):

    max_depth_range = list(range(1, 10))
    max_depth_range.append(str("None"))

    for depth in max_depth_range:
        if(depth == "None"):
            clf = RandomForestClassifier(random_state = 0)
            clf.fit(x_train, y_train)
        else:
            clf = RandomForestClassifier(max_depth = depth, random_state = 0)
            clf.fit(x_train, y_train)

        accuracy = clf.score(x_test, y_test_stratified)*100
        print("Depth: ", depth, " Accuracy: ", accuracy)
```

Figure 12: Depth Code

After comparing our result with base model, as we can see, when the depth become higher, we can catch the base model. For the last model, I can use ***7, 8, 9***

```
Stratified One Hold Out Fold
Random Forest  Accuracy:  0.975


Depth:  1  Accuracy:  92.5
Depth:  2  Accuracy:  96.25
Depth:  3  Accuracy:  96.25
Depth:  4  Accuracy:  96.25
Depth:  5  Accuracy:  96.25
Depth:  6  Accuracy:  96.25
Depth:  7  Accuracy:  97.5
Depth:  8  Accuracy:  97.5
Depth:  9  Accuracy:  97.5
Depth:  None  Accuracy:  97.5
```

Figure 13: Depth Outcome

I continue to my experience with split strategy using **min_samples_split** and **criterion**. To do that I used code below.

```
def tuningSplit(x_train, x_test, y_train, y_test_stratified):
    criterion = ["gini", "entropy"]

    for i in criterion:
        clf = RandomForestClassifier(criterion = i, max_depth = 7, random_state = 0)
        clf.fit(x_train, y_train)

        accuracy = clf.score(x_test, y_test_stratified)*100
        print("Criterion: ", i, "   Accuracy: ", accuracy)

    for i in range(2, 10):
        clf = RandomForestClassifier(max_depth = 7, min_samples_split = i, random_state = 0)
        clf.fit(x_train, y_train)

        accuracy = clf.score(x_test, y_test_stratified)*100
        print("min_samples_split: ", i, "   Accuracy: ", accuracy)
```

Figure 14: Split Code

After comparing our result with base model, as we can see, when the **min_samples_split** become higher, we decreased our accuracy, and **criterion** did not change anything. For the last model, I can use **min_samples_split** as ***2 or 3***

```
Stratified One Hold Out Fold
Random Forest  Accuracy:  0.975


Criterion:  gini    Accuracy:  97.5
Criterion:  entropy    Accuracy:  97.5
min_samples_split:  2    Accuracy:  97.5
min_samples_split:  3    Accuracy:  97.5
min_samples_split:  4    Accuracy:  96.25
min_samples_split:  5    Accuracy:  96.25
min_samples_split:  6    Accuracy:  96.25
min_samples_split:  7    Accuracy:  96.25
min_samples_split:  8    Accuracy:  96.25
min_samples_split:  9    Accuracy:  96.25
```

Figure 15: Split Outcome

Lastly, individually, we changed the class_weight parameter. To do that, I used the code below.

```
def tuningClassWeight(x_train, x_test, y_train, y_test_stratified):
    # No class weight
    clf = RandomForestClassifier(max_depth = 7, random_state = 0)
    clf.fit(x_train, y_train)

    accuracy = clf.score(x_test, y_test_stratified)*100
    print("Class weight: None           Accuracy: ", accuracy)

    # Balanced class weight
    clf = RandomForestClassifier(max_depth = 7, random_state = 0, class_weight ='balanced')
    clf.fit(x_train, y_train)

    accuracy = clf.score(x_test, y_test_stratified)*100
    print("Class weight: Balanced       Accuracy: ", accuracy)
```

Figure 16: Class weight Code

After comparing our result with base model, as we can see, when the **class_weight** does not change anything.

```
Stratified One Hold Out Fold
Random Forest  Accuracy:  0.975


Class weight: None              Accuracy:  97.5
Class weight: Balanced          Accuracy:  97.5
```

Figure 17: Class Weight Outcome

In order to decide the final model, we conduct one more experiment. We combined all approaches mention above. This experiment happened inside **display-Accuracy** method After finding the best model, lastly, I tried to find optimal **n_estimator number**. Since the results are too much, I did not create a table, but I chose these parameters which are n_estimator = 100, criterion = entropy, max_depth = 7, min_samples_split = 3 and class_weight = "balanced" to use in final model.

To check the correctness of the model, I checked the four errors written below:

```
def fourError(X, Y, model):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=0, strat
ify=Y)

    Train_x, TrainDev_x, Train_y, TrainDev_y = train_test_split(X_train, Y_train, test_size=0.2, r
andom_state=0, stratify=Y_train)
    Dev_x, Test_x, Dev_y, Test_y = train_test_split(X_test, Y_test, test_size=0.5, random_state=0,
stratify=Y_test)

    model.fit(Train_x, Train_y)

    y_true, trainDev_pred = TrainDev_y, model.predict(TrainDev_x)

    print("Train-Train Dev,    e1:", metrics.mean_squared_error(TrainDev_y, trainDev_pred),"\n")
    print("Accuracy: ", 1 - metrics.mean_squared_error(TrainDev_y, trainDev_pred))
    print( '\nClassification report\n' )
    print(classification_report(y_true, trainDev_pred))

    y_true, dev_pred = Dev_y, model.predict(Dev_x)
    print("Train-Dev,    e2", metrics.mean_squared_error(Dev_y, dev_pred),"\n")
    print("Accuracy: ", 1 - metrics.mean_squared_error(Dev_y, dev_pred))
    print( '\nClassification report\n' )
    print(classification_report(y_true, dev_pred))

    y_true, test_pred = Test_y, model.predict(Test_x)
    print("Train-Test,    e3: ", metrics.mean_squared_error(Test_y, test_pred),"\n")
    print("Accuracy: ", 1 - metrics.mean_squared_error(Test_y, test_pred))
    print( '\nClassification report\n' )
    print(classification_report(y_true, test_pred))

    y_true, devTest_pred = Y_test, model.predict(X_test)
    print("Train-(Dev+Test),    e4: ", metrics.mean_squared_error(Y_test, devTest_pred),"\n")
    print("Accuracy: ", 1 - metrics.mean_squared_error(Y_test, devTest_pred))
    print( '\nClassification report\n' )
    print(classification_report(y_true, devTest_pred))
```

Figure 18: Four error

I tested the model with raw and cleaned data, and I saw that the results are okay, but the results are very inconsistent.

Train-Train Dev e1	**Train-Dev e2**	**Train-Test e3**	**Train-(Dev+Test) e4**
0.95	0.96470	0.91860	0.94152

Table 1: Four errors for raw data

Train-Train Dev e1	**Train-Dev e2**	**Train-Test e3**	**Train-(Dev+Test) e4**
0.925	0.98823	0.89534	0.94152

Table 2: Four errors for cleaned data

4 Support Vector Machine

The objective of the support vector machine algorithm is to find a hyper plane in an N-dimensional space(N — the number of features) that distinctly classifies the data points. To separate the two classes of data points, there are many possible hyper planes that could be chosen.
First, we divided our data 70 percent as train data and 30 percent as test data. Then, we used different validation techniques on our train and test data to see which one is given the best result among them. For this purpose, we created a method called SVM.

4.1 Experiments

To do our experiment, we created a method called SVM. we will apply our validation methods as well.

```python
def svm(X_train, Y_train, kernel, weight, gamma):

    svm = SVC(C=1, kernel=kernel, degree=3, gamma=gamma, coef0=0.0, shrinking=True,
          probability=False, tol=0.001, cache_size=200, class_weight=weight,
          max_iter=-1, decision_function_shape="ovr", random_state = 0)

    #5-Fold
    cv_result_svm_5 = cross_val_score(svm, X_train, Y_train, cv=5, scoring='accuracy')

    #10-Fold
    cv_result_svm_10 = cross_val_score(svm, X_train, Y_train, cv=10, scoring='accuracy')

    #Random One Holdout
    x_train, x_test, y_train, y_test_random = randomOneHoldout(X_train, Y_train)
    svm.fit(x_train, y_train)
    y_pred_svm_random = svm.predict(x_test)

    #Stratified One Holdout
    x_train, x_test, y_train, y_test_stratified = stratifiedOneHoldout(X_train, Y_train)
    svm.fit(x_train, y_train)
    y_pred_svm_stratified = svm.predict(x_test)

    print("5 Fold")
    print("SVM Accuracy: ", cv_result_svm_5.mean())

    print("10 Fold")
    print("SVM Accuracy: ", cv_result_svm_10.mean())

    print("Random One Hold Out")
    print("SVM Accuracy: ", 1 - metrics.mean_squared_error(y_test_random, y_pred_svm_random))

    print("Stratified One Hold Out Fold")
    print("SVM Accuracy: ", 1 - metrics.mean_squared_error(y_test_stratified, y_pred_svm_stratifie
d))
```

Figure 19: SVM code

We generally look our SVM model's result below, and we decided to move on with Stratified One Hold Out because it give higher accuracy. After we decided to go with Stratified One Hold Out , we started to feed our method with Kernel parameters. We used four different kernel parameters which are Linear, Poly, Rbf, Sigmoid.

```
5 Fold
SVM Accuracy:  0.8994936708860759
10 Fold
SVM Accuracy:  0.902051282051282
Random One Hold Out
SVM Accuracy:  0.9625
Stratified One Hold Out Fold
SVM Accuracy:  0.9625
```

Figure 20: SVM Base Model Result

As we can see, we got the worst result by Sigmoid, and we got the best result by Linear. We continued with Linear. Then, we continued with class weight. In class_weight parameter, we have two option which are None and Balanced.

Because of the problem with colabs, I could not paste the results, but our result, 0.9875, stay same when we apple None parameter, but decreased when we applied Balanced. Finally, we made changes on gamma value. During our experiment, we figured out that Linear kernel only works with when gamma value ”auto”, and the result stayed same. As a result, we got quite good results, but I order to validate it, we need to look four error result. Also, I tried the model with cleaned data as well.

Train-Train Dev e1	**Train-Dev e2**	**Train-Test e3**	**Train-(Dev+Test) e4**
0.97802	0.94737	0.94737	0.94737

Table 3: Four errors for raw data

Train-Train Dev e1	**Train-Dev e2**	**Train-Test e3**	**Train-(Dev+Test) e4**
0.97802	0.94737	0.94737	0.94737

Table 4: Four errors for cleaned data

We got really good results, and they are most stable ones by far. We can apply boosting and bagging to final models.

5 Boosting

Boosting is one of most famous approaches and it produces an ensemble model that is in general less biased than the weak learners that compose it. Boosting methods work in the same spirit as bagging methods. We build a family of models that are aggregated to obtain a strong learner that performs better. However, unlike bagging that mainly aims at reducing variance, boosting is a technique that consists in fitting sequentially multiple weak learners in a very adaptative way.

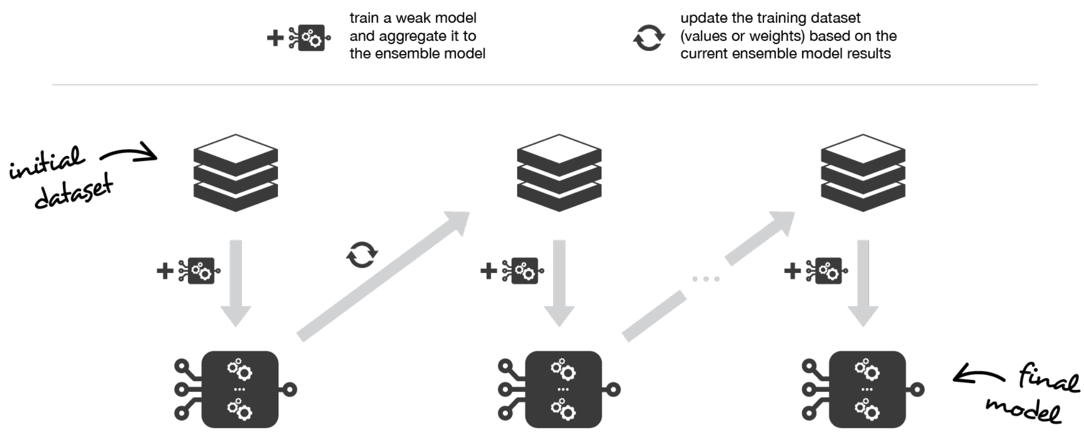

Figure 21: Boosting

Bagging, that often considers homogeneous weak learners, learns them independently from each other in parallel and combines them following some kind of deterministic averaging process.

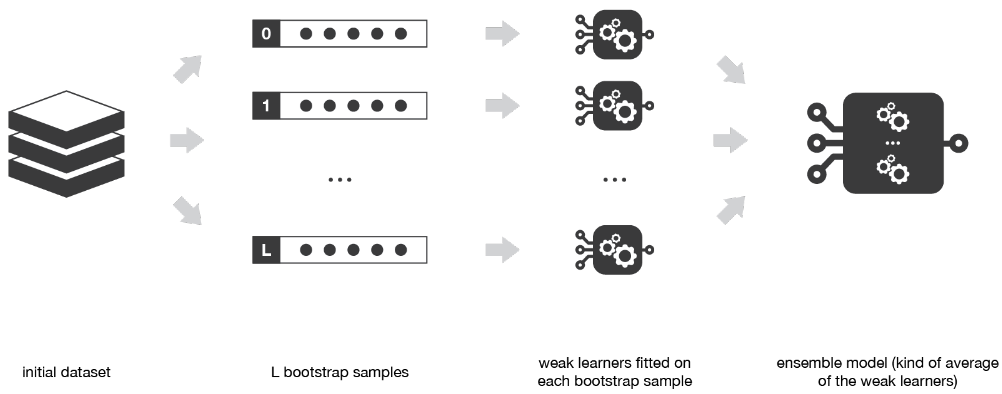

Figure 22: Bagging

5.1 Adaboost

In adaboosting (often called Adaptive Boost), we try to define our ensemble model as a weighted sum of L weak learners. In sklearn, we can feed our models to built-in adaboost method and we can get result.

$$s_L(.) = \sum_{l=1}^{L} c_l \times w_l(.) \qquad \text{where } c_l\text{'s are coefficients and } w_l\text{'s are weak learners}$$

Figure 23: Adaboost Formula

5.2 Random Forest

In bagging, the random forest is one of the techniques for ensemble learning. The random forest approach where deep trees, fitted on bootstrap samples, are combined to produce an output with lower variance. However, random forests also use another trick to make the multiple fitted trees a bit less correlated with each others.

5.3 Experiment

For the experiment, we chose ***SVM*** comparing our results. We will apply boost and bagging methods to the both model. For boosting, we will compare it and change their two parameters which are learning_rate and n_estimators.

```
def AdaBoost(model, n_estimators, learning_rate, X_train, Y_train, X_test, Y_test):
    clf = AdaBoostClassifier(base_estimator = model, n_estimators= n_estimators, learning_rate=lea
rning_rate, random_state=0)
    clf.fit(X_train, Y_train)
    clf.predict(X_test)
    return clf.score(X_train, Y_train)
```

```
def GradientBoost(n_estimators, learning_rate, X_train, Y_train, X_test, Y_test):
    clf = GradientBoostingClassifier(n_estimators= n_estimators, learning_rate=learning_rate, rand
om_state=0)
    clf.fit(X_train, Y_train)
    clf.predict(X_test)
    return clf.score(X_train, Y_train)
```

Figure 24: Adaboost and Gradient Boost

For boosting, when the learners increased, we got lower accuracy. Also, when we changed the estimators, it did not affect the result. Overall, boosting increased our accuracy.

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=0)
#best model

learning_rate = [0.0001, 0.001, 0.01, 0.1, 1 ,2 ,3 ,4, 5]

result_ada_2 = []
for i in learning_rate: #i -> Learning Rate
    x = []
    for j in range(50, 150, 25): #j -> N estimators
        x.append(AdaBoost(svm, j, i, X_train, Y_train, X_test, Y_test))
    result_ada_2.append(x)
```

```
x = pd.DataFrame(result_ada_2)
x
```

	0	1	2	3
0	0.956044	0.956044	0.956044	0.956044
1	0.956044	0.956044	0.953846	0.953846
2	0.951648	0.951648	0.951648	0.951648
3	0.951648	0.951648	0.942857	0.934066
4	0.742857	0.701099	0.663736	0.650549
5	0.912088	0.909890	0.905495	0.907692
6	0.914286	0.865934	0.786813	0.720879
7	0.756044	0.646154	0.639560	0.637363
8	0.648352	0.637363	0.637363	0.637363

Figure 25: Boosting Result

For bagging, I only changed estimator number, and bagging gave pretty bad results.

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=0)
#best model


result_bagging = []
for j in range(50, 150, 25): #j -> N estimators
    result_bagging.append(bagging(svm, j, X_train, Y_train, X_test, Y_test))

```

```
y = pd.DataFrame(result_bagging)
y
```

	0
0	0.736181
1	0.695980
2	0.665829
3	0.650754

Figure 26: Bagging Result

5.4 Learning Rate and N Estimators

Learning rate means that determines how much weak learners contribute to the weight of each iteration. Decreasing the learning rate makes the coefficients smaller, which reduces the amplitude of the sample_weights at each step. This translates into:

- Smaller variations of the weighted data points
- Fewer differences between the weak classifier decision boundaries

N estimators means that number of weak learners to train iteratively. Increasing the number of weak classifiers, increases the number of iterations, and allows the sample weights to gain greater amplitude. This translates into:

- More weak classifiers to combine at the end
- More variations in the decision boundaries of these classifiers

6 Appendix

6.1 Project Link

It will available after exam. `https://github.com/kaankoken/random_forest_vs_svm`

6.2 Code

```python
# -*- coding: utf-8 -*-
"""svm-vs-random-forest.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/github/kaankoken/random_forest_vs_svm/blob/master/svm_vs_random_forest.ipynb
"""

from sklearn import metrics, datasets, preprocessing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.model_selection import train_test_split, KFold, StratifiedKFold, cross_val_score

from sklearn.ensemble import RandomForestClassifier, BaggingClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.svm import LinearSVC, SVC

from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import classification_report

def randomOneHoldout(X_train, Y_train):
    x_train, x_test, y_train, y_test = train_test_split(X_train, Y_train, test_size=0.2, random_state=0)
    return x_train, x_test, y_train, y_test

def stratifiedOneHoldout(X_train, Y_train):
    x_train, x_test, y_train, y_test = train_test_split(X_train, Y_train, test_size=0.2, random_state=0)
    return x_train, x_test, y_train, y_test
```

```
def visulize_class(df):
  # look at the last column on data frame (the classification value
                                 column)
  df.iloc[:, -1].value_counts().plot(kind='bar')
  plt.title("Output Distribution: Breast Cancer DS")
  plt.xlabel("Classification")
  plt.ylabel("Frequency")
  plt.show()

def identify_noise(df):

  noise = df[df.isnull().any(axis=1)].count()
  total_noise = noise.sum()
  print("{0} null values were found.".format(str(total_noise)))
  if(total_noise > 0):
    print(noise)
  print("\n\nShowing all data types:\n\n")
  print(df.dtypes)

def heat_map(df):
  fig, ax = plt.subplots()
  corr = df.corr()
  sns.heatmap(corr, annot=True, cmap='hot')
  plt.show()

def filter_features(data, bad_indices):
  # eliminate above column indices from the data and return new set
  filtered_data = np.delete(data, bad_indices, axis=1)

  return filtered_data

def vis_all_feat(data, class_):
    for col_ind in range(data.shape[1]):
        print("Viewing Feature #{0}".format(str(col_ind)))
        vis_single_feat(data, class_, col_ind)

def vis_single_feat(data, class_, ind):
  # create graph of classification and feature values
  plt.figure(100) # display two plots on separate figures
  df = pd.DataFrame(data)
  feat_vals = df.iloc[:, ind]
  plt.scatter(feat_vals, class_)
  plt.title("Plot of Feature {0}".format(str(ind)))
  plt.xlabel("Feature Value")
  plt.ylabel("Classification")

  # create bar graph of mean feature values for each classification
  plt.figure(200)
  plt.title("Mean Values of Feature {0}".format(str(ind)))
  plt.xlabel("Classification")
```

```
    plt.ylabel("Mean Feature Value")
    mean_df = pd.concat([df.iloc[:, ind], pd.Series(class_)], axis=1)
    mean_df.columns = ["values", "classif"]
    mean_df.groupby("classif", as_index=False)["values"].mean().loc[:
                                ,"values"].plot(kind='bar')

    plt.show()

def plot_pairplot(data, class_):
    data_df = pd.DataFrame(data)
    # add classification so the plot can be colored by it
    data_df.loc[:, "classif"] = pd.Series(class_)
    sns.pairplot(data_df, hue='classif')
    plt.show()

def fourError(X, Y, model):
     X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                test_size=0.2, random_state=0,
                                stratify=Y)

     Train_x, TrainDev_x, Train_y, TrainDev_y = train_test_split(
                                X_train, Y_train, test_size=0.2,
                                random_state=0, stratify=Y_train)
     Dev_x, Test_x, Dev_y, Test_y = train_test_split(X_test, Y_test,
                                 test_size=0.5, random_state=0,
                                stratify=Y_test)

     model.fit(Train_x, Train_y)

     y_true, trainDev_pred = TrainDev_y, model.predict(TrainDev_x)

     print("Train-Train Dev,   e1:", metrics.mean_squared_error(
                                TrainDev_y, trainDev_pred),"\n")
     print("Accuracy: ", 1 - metrics.mean_squared_error(TrainDev_y,
                                trainDev_pred))
     print( '\nClassification report\n' )
     print(classification_report(y_true, trainDev_pred))

     y_true, dev_pred = Dev_y, model.predict(Dev_x)
     print("Train-Dev,   e2", metrics.mean_squared_error(Dev_y,
                                dev_pred),"\n")
     print("Accuracy: ", 1 - metrics.mean_squared_error(Dev_y,
                                dev_pred))
     print( '\nClassification report\n' )
     print(classification_report(y_true, dev_pred))

     y_true, test_pred = Test_y, model.predict(Test_x)
     print("Train-Test,   e3: ", metrics.mean_squared_error(Test_y,
                                test_pred),"\n")
```

```
    print("Accuracy: ", 1 - metrics.mean_squared_error(Test_y, test_pred))
    print( '\nClassification report\n' )
    print(classification_report(y_true, test_pred))

    y_true, devTest_pred = Y_test, model.predict(X_test)
    print("Train-(Dev+Test),   e4: ", metrics.mean_squared_error(Y_test, devTest_pred),"\n")
    print("Accuracy: ", 1 - metrics.mean_squared_error(Y_test, devTest_pred))
    print( '\nClassification report\n' )
    print(classification_report(y_true, devTest_pred))

def svm(X_train, Y_train, kernel, weight, gamma):

    svm = SVC(C=1, kernel=kernel, degree=3, gamma=gamma, coef0=0.0, shrinking=True,
          probability=False, tol=0.001, cache_size=200, class_weight=weight,
          max_iter=-1, decision_function_shape="ovr", random_state = 0)

    #5-Fold
    cv_result_svm_5 = cross_val_score(svm, X_train, Y_train, cv=5, scoring='accuracy')

    #10-Fold
    cv_result_svm_10 = cross_val_score(svm, X_train, Y_train, cv=10, scoring='accuracy')

    #Random One Holdout
    x_train, x_test, y_train, y_test_random = randomOneHoldout(X_train, Y_train)
    svm.fit(x_train, y_train)
    y_pred_svm_random = svm.predict(x_test)

    #Stratified One Holdout
    x_train, x_test, y_train, y_test_stratified = stratifiedOneHoldout(X_train, Y_train)
    svm.fit(x_train, y_train)
    y_pred_svm_stratified = svm.predict(x_test)

    print("5 Fold")
    print("SVM Accuracy: ", cv_result_svm_5.mean())

    print("10 Fold")
    print("SVM Accuracy: ", cv_result_svm_10.mean())

    print("Random One Hold Out")
```

```python
    print("SVM Accuracy: ", 1 - metrics.mean_squared_error(y_test_random, y_pred_svm_random))

    print("Stratified One Hold Out Fold")
    print("SVM Accuracy: ", 1 - metrics.mean_squared_error(y_test_stratified, y_pred_svm_stratified))

def random_forest(X_train, Y_train, forest_size, max_depth, criterion, min_samples_split, class_weight):
    rf = RandomForestClassifier(n_estimators=forest_size, oob_score=True, n_jobs=-1, max_depth=max_depth, criterion=criterion, min_samples_split = min_samples_split, class_weight=class_weight, random_state=0)

    #5-Fold
    cv_result_rf_5 = cross_val_score(rf, X_train, Y_train, cv=5, scoring='accuracy')

    #10-Fold
    cv_result_rf_10 = cross_val_score(rf, X_train, Y_train, cv=10, scoring='accuracy')

    #Random One Holdout
    x_train, x_test, y_train, y_test_random = randomOneHoldout(X_train, Y_train)
    rf.fit(x_train, y_train)
    y_pred_rf_random = rf.predict(x_test)

    #Stratified One Holdout
    x_train, x_test, y_train, y_test_stratified = stratifiedOneHoldout(X_train, Y_train)
    rf.fit(x_train, y_train)
    y_pred_rf_stratified = rf.predict(x_test)

    print("Stratified One Hold Out Fold")
    print("Random Forest  Accuracy: ", 1 - metrics.mean_squared_error(y_test_stratified, y_pred_rf_stratified))

    print("5 Fold")
    print("Random Forest Accuracy: ", cv_result_rf_5.mean())

    print("10 Fold")
```

```
    print("Random Forest  Accuracy: ", cv_result_rf_10.mean())

    print("Random One Hold Out")
    print("Random Forest  Accuracy: ", 1 - metrics.mean_squared_error(y_test_random, y_pred_rf_random))

def tuningDepth(x_train, x_test, y_train, y_test_stratified):

    max_depth_range = list(range(1, 10))
    max_depth_range.append(str("None"))

    for depth in max_depth_range:
        if(depth == "None"):
            clf = RandomForestClassifier(random_state = 0)
            clf.fit(x_train, y_train)
        else:
            clf = RandomForestClassifier(max_depth = depth, random_state = 0)
            clf.fit(x_train, y_train)

        accuracy = clf.score(x_test, y_test_stratified)*100
        print("Depth: ", depth, " Accuracy: ", accuracy)

def tuningSplit(x_train, x_test, y_train, y_test_stratified):
    criterion = ["gini", "entropy"]

    for i in criterion:
        clf = RandomForestClassifier(criterion = i, max_depth = 7, random_state = 0)
        clf.fit(x_train, y_train)

        accuracy = clf.score(x_test, y_test_stratified)*100
        print("Criterion: ", i, "   Accuracy: ", accuracy)

    for i in range(2, 10):
        clf = RandomForestClassifier(max_depth = 7, min_samples_split = i, random_state = 0)
        clf.fit(x_train, y_train)

        accuracy = clf.score(x_test, y_test_stratified)*100
        print("min_samples_split: ", i, "   Accuracy: ", accuracy)

def tuningClassWeight(x_train, x_test, y_train, y_test_stratified):
    # No class weight
    clf = RandomForestClassifier(max_depth = 7, random_state = 0)
    clf.fit(x_train, y_train)

    accuracy = clf.score(x_test, y_test_stratified)*100
```

```
    print("Class weight: None           Accuracy: ", accuracy)

    # Balanced class weight
    clf = RandomForestClassifier(max_depth = 7, random_state = 0,
                                  class_weight ='balanced')
    clf.fit(x_train, y_train)

    accuracy = clf.score(x_test, y_test_stratified)*100
    print("Class weight: Balanced       Accuracy: ", accuracy)

def AdaBoost(model, n_estimators, learning_rate, X_train, Y_train,
                                  X_test, Y_test):
    clf = BaggingClassifier(base_estimator = model, n_estimators=
                                  n_estimators, learning_rate=
                                  learning_rate, random_state=0)
    clf.fit(X_train, Y_train)
    clf.predict(X_test)
    return clf.score(X_train, Y_train)

def bagging(model, n_estimators, X_train, Y_train, X_test, Y_test):
    clf = AdaBoostClassifier(base_estimator = model, n_estimators=
                                  n_estimators, random_state=0)
    clf.fit(X_train, Y_train)
    clf.predict(X_test)
    return clf.score(X_train, Y_train)

def displayAccuracy(X, Y):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                  test_size=0.3, random_state=0)

    kernel = ["linear", "rbf", "poly", "sigmoid"]
    weight = [None, "balanced"]
    gamma = ["auto", "scale"]

    for i in kernel:
        for j in weight:
            for k in gamma:
                if i != "linear":
                    print("Kernel: {} - Weight: {} - Gamma: {}".
                                  format(str(i), j, k))
                    svm(X_train, Y_train, i, j, k)
                else:
                    print("Kernel: {} - Weight: {} - Gamma: {}".
                                  format(str(i), j, "auto"))
                    svm(X_train, Y_train, i, j, k)
        print()

    combined approach
    for depth in max_depth_range:
        for c in criterion:
```

```
            for i in range(2, 10):
                for w in weight:
                    print("Depth: {}, Criterion: {}, Min Split {},
                                  Weight: {}".format(depth, c, i, w
                                  ))
                    random_forest(X_train, Y_train, 100, depth, c,
                                  i, w)
        print("\n")

    forest_size = [25, 50, 75, 100, 125, 150, 175, 200]

    for s in forest_size:
        random_forest(X_train, Y_train, s, 7, "entropy", 3, "
                                  balanced")

def compareBaseModels(X, Y):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                  test_size=0.3, random_state=0)

    svm(X_train, Y_train, "rbf", None, "scale")
    print("")
    random_forest(X_train, Y_train, 100, None, "gini", 2, None)

    x_train, x_test, y_train, y_test_stratified =
                                  stratifiedOneHoldout(X_train,
                                  Y_train)
    #Individual tuning
    print("\n")
    tuningDepth(x_train, x_test, y_train, y_test_stratified);
    tuningSplit(x_train, x_test, y_train, y_test_stratified)
    tuningClassWeight( x_train, x_test, y_train, y_test_stratified)

if __name__ == '__main__':

    breast_cancer = datasets.load_breast_cancer()
    X = breast_cancer.data
    Y = breast_cancer.target

    #Shape of the data
    print(X.shape, end="\n")

    feauture = pd.DataFrame(Y)
    df = pd.DataFrame(X)
    rf = RandomForestClassifier(n_estimators=100, oob_score=True,
                                  n_jobs=-1, max_depth=7, criterion
                                  ="entropy", min_samples_split = 3
                                  , class_weight="balanced",
                                  random_state=0)
    svm = SVC(C=1, kernel="linear", degree=3, gamma="scale", coef0=
                                  0.0, shrinking=True,
```

```
            probability=True, tol=0.001, cache_size=200, class_weight
                                                =None,
            max_iter=-1, decision_function_shape="ovr", random_state
                                                = 0)
    fourError(X, Y, svm)
    visulize_class(feauture)
    compareBaseModels(X, Y)
    displayAccuracy(X, Y)

    identify_noise(df)
    heat_map(df)

    #filter strongly correlated features - can see which ones in
                                                correlation map
    X = filter_features(X, [2, 3, 20, 22, 23, 12, 13])

    vis_all_feat(X, Y)
    X = filter_features(X, [1, 2, 6, 7, 9, 10, 14, 15])
    print("Cleaned data")
    fourError(X, Y, svm)
    #remaining features
    plot_pairplot(X, Y)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size
                                                =0.3, random_state=0)
#best model

learning_rate = [0.0001, 0.001, 0.01, 0.1, 1 ,2 ,3 ,4, 5]

result_ada_2 = []
for i in learning_rate: #i -> Learning Rate
    x = []
    for j in range(50, 150, 25): #j -> N estimators
        x.append(AdaBoost(svm, j, i, X_train, Y_train, X_test,
                                                Y_test))
    result_ada_2.append(x)

x = pd.DataFrame(result_ada_2)
x

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size
                                                =0.3, random_state=0)
#best model


result_bagging = []
for j in range(50, 150, 25): #j -> N estimators
    result_bagging.append(bagging(svm, j, X_train, Y_train, X_test,
                                                Y_test))

```

```
y = pd.DataFrame(result_bagging)
y
```