MIDDLE EAST TECHNICAL UNIVERSITY

NORTHERN CYPRUS CAMPUS

# CNG 562
# MACHINE LEARNING

ASSIGNMENT-2

# Report

*Nisa Nur Odabaş*
*Kaan Taha Köken*

June 2, 2020

# Contents

# List of Figures

# 1 Introduction

In this assignment, our aim is to combine methodologies learned earlier such as preprocessing and validation techniques with new classification methods, which are K-Nearest Neighbor (KNN), Naïve Bayes, Support Vector Machine (SVM) and Decision Tree. We are also aiming to find the best and worst classification method among them and try to use boosting methods on the best and worst classifier.

## 1.1 Dataset

We are using Iris dataset for the assignment. It contains 3 classes of 50 instances each, where each class refers a type of iris plant. It also has four attributes.
Attributes:

- sepal length(cm)
- sepal width(cm)
- petal length(cm).
- petal width(cm)
- class
  - Iris Setosa
  - Iris Versicolour
  - Iris Virginica

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

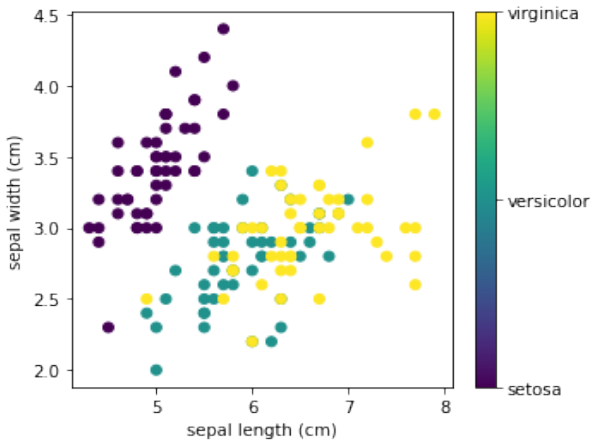


Figure 1: Data visualization

Also, if we remove one of the features using Principal Component Analysis (PCA) method, our data look like in 3 Dimension.

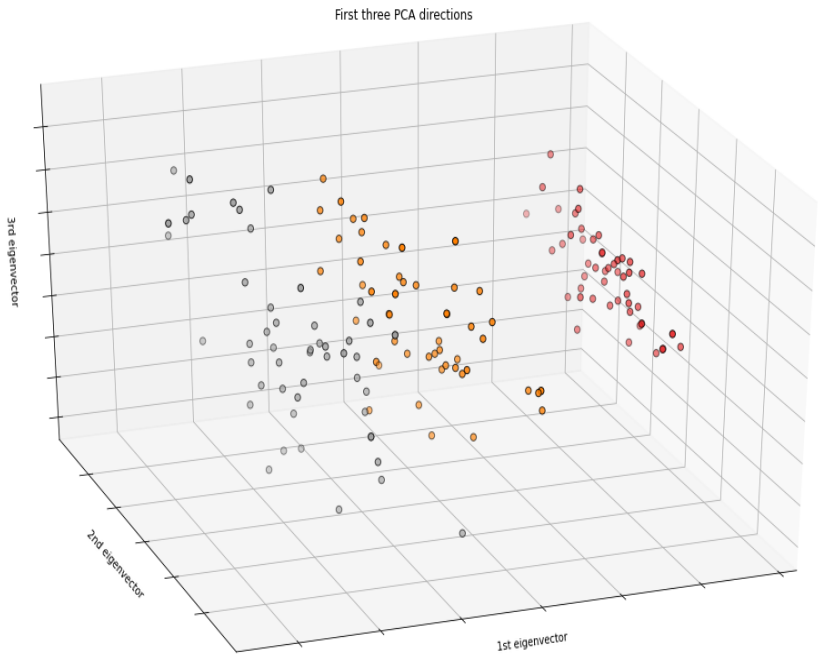


Figure 2: Data visualization in 3D

## 1.2 Preprocess

### 1.2.1 Z-Score

Z-score normalization is a technique to normalize data and solves outlier issue. Basically, it gives an idea of how far from the mean a data point is.

$$\frac{value - \mu}{\sigma}$$

Figure 3: Z-Score formula

$\mu$ is the mean value of the feature and $\sigma$ is the standard deviation of the feature. If a value is exactly equal to the mean of all the values of the feature, it will be normalized to 0.

## 1.3 Validation

### 1.3.1 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.

### 1.3.2 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.

### 1.3.3 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.

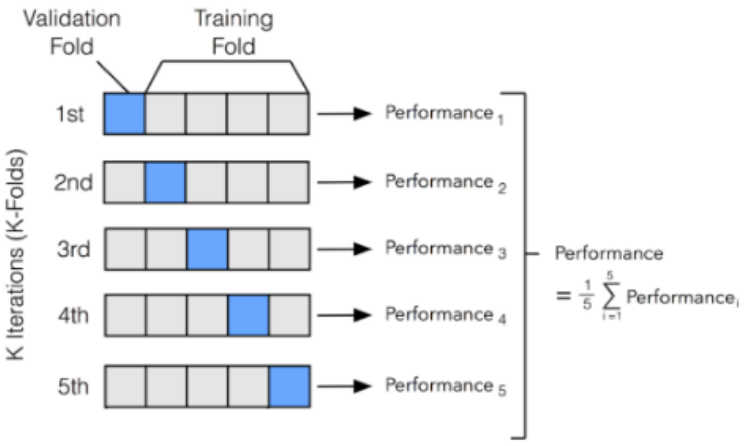


Figure 4: 5-Fold

# 2 K-Nearest Neighbors

The k-nearest neighbors (KNN) algorithm is a simple supervised machine learning algorithm that can be used to solve both classification and regression problems. In our experiment, we will use raw data and normalized data by Z-score normalization. In both approach, we will chance K values and distance metrics, and see which will give the best result.

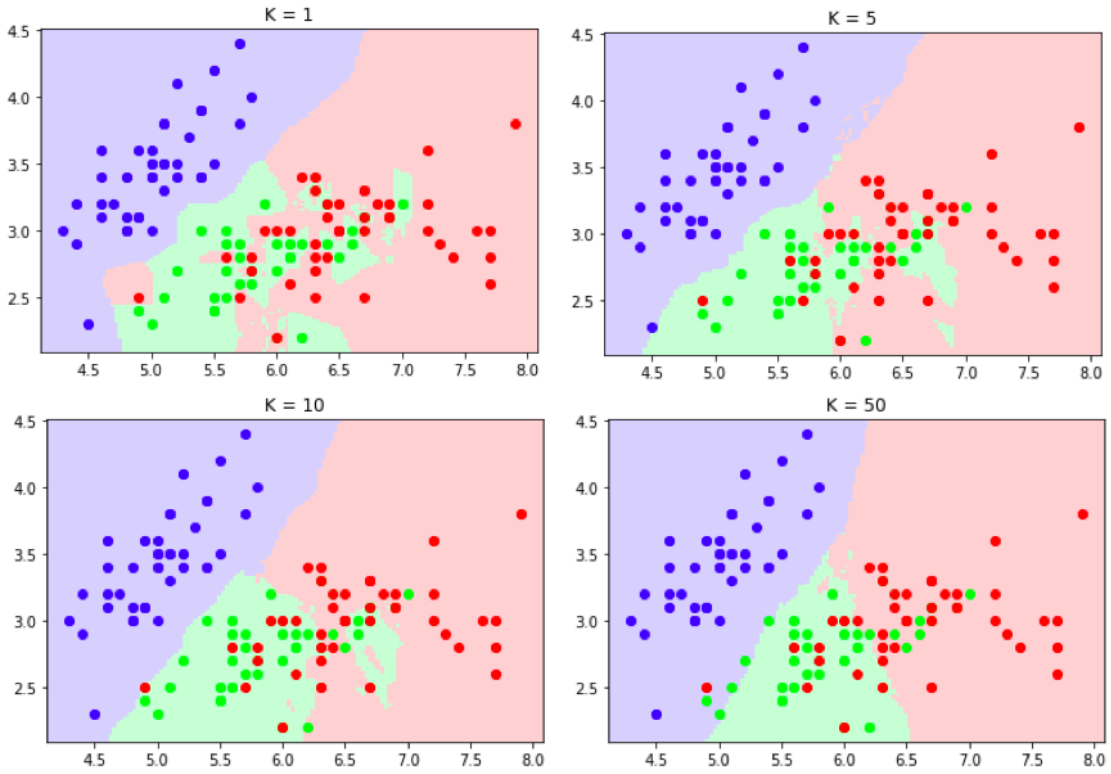


Figure 5: KNN with different K

## 2.1 Distance Metrics

In KNN predictor, we have a lot distance metrics to use in. Through over our experiment, we will use different metrics and we will choose best result provider.

### 2.1.1 Minkownski Distance

Minkowski distance is a metric in Normed vector space. A Normed vector space is a vector space on which a norm is defined.

$$\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$$

Figure 6: Formula of Minkowski Distance

Minkowski distance is the generalized distance metric. We can manipulate the above formula to calculate the distance between two data points in different ways. As mentioned above, we can manipulate the value of ***p*** and calculate the distance in three different ways:

- p = 1, Manhattan Distance
- p = 2, Euclidean Distance
- p = $\infty$, Chebychev Distance.

### 2.1.2 Manhattan Distance

In Manhattan distance, if we need to calculate the distance between two data points in a grid like path. As mentioned above, we use ***Minkowski*** distance formula to find Manhattan distance by setting ***p***'s value as ***1***. Distance d will be calculated using an ***absolute sum of difference*** between its cartesian co-ordinates as below:

$$d = \sum_{i=1}^{n} |x_i - y_i|$$

Figure 7: Formula of Manhattan Distance

### 2.1.3 Euclidean Distance

Euclidean distance is one of the most used distance metric. It is calculated using Minkowski Distance formula by setting $\boldsymbol{p}$'s value to $\boldsymbol{2}$.

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Figure 8: Formula of Euclidean Distance

### 2.1.4 Mahalonobis Distance

Mahalonobis distance is the distance between a point and a distribution. And not between two distinct points. It is effectively a multivariate equivalent of the Euclidean distance.

$$D^2 = (x-m)^T \cdot C^{-1} \cdot (x-m)$$

Figure 9: Formula of Mahalonobis Distance

### 2.1.5 Chebyshev Distance

Chebyshev distance is also called Maximum value distance. It examines the absolute magnitude of the differences between coordinates of a pair of objects.

$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

Figure 10: Formula of Chebyshev Distance

## 2.2 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 kNN.

```
def kNN(k: int, metric: str, X_train, Y_train):

    #Model
    if metric == "mahalanobis":
      knn = KNeighborsClassifier(n_neighbors=k, weights='uniform', metric=metric, algorithm="brute
", metric_params={'V': np.cov(X_train)})
    else:
      knn = KNeighborsClassifier(n_neighbors=k, weights='uniform', metric=metric)

    #5-Fold
    cv_result_knn_5 = cross_val_score(knn, X_train, Y_train, cv=5, scoring='accuracy')

    #10-Fold
    cv_result_knn_10 = cross_val_score(knn, 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)
    knn.fit(x_train, y_train)

    y_pred_knn_random = knn.predict(x_test)

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

    print("5 Fold")
    print("KNN Accuracy: ", cv_result_knn_5.mean())

    print("10 Fold")
    print("KNN Accuracy: ", cv_result_knn_10.mean())

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

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

Figure 11: KNN Code

| | **K = 3** | **K = 5** | **K = 7** | **K = 9** | **K = 11** |
|---|---|---|---|---|---|
| **5 Fold** | 0.925 | 0.93333 | 0.95 | 0.95833 | 0.95833 |
| **10 Fold** | 0.925 | 0.93333 | 0.95 | 0.95833 | 0.95833 |
| **Random One Hold Out** | 0.91667 | 0.91667 | 0.91667 | 0.91667 | 0.91667 |
| **Stratified One Hold Out** | 0.91667 | 0.91667 | 0.91667 | 0.91667 | 0.91667 |

Table 1: Raw Data with Different K values

| | **K = 3** | **K = 5** | **K = 7** | **K = 9** | **K = 11** |
|---|---|---|---|---|---|
| **5 Fold** | 0.925 | 0.93333 | 0.94167 | 0.93333 | 0.925 |
| **10 Fold** | 0.93333 | 0.93333 | 0.95 | 0.93333 | 0.93333 |
| **Random One Hold Out** | 0.875 | 0.875 | 0.875 | 0.91667 | 0.91667 |
| **Stratified One Hold Out** | 0.875 | 0.875 | 0.875 | 0.91667 | 0.91667 |

Table 2: Z-Score Normalized Data with Different K values

After our little experiments, we saw our result, and for both raw and z-score normalized data, we decided go with 5 or 10 Fold since the others gave low accuracy.

We continue with 10-Fold and after that we started to experiment on both Raw and Z-score normalized data while changing the distance metrics. In order to see the data, we created tables. For raw data, it is clear that when $\boldsymbol{K = 7}$ and the distance metric is ***Chebyshev***, we got best accuracy. In order to prove the result is correct or not, we will use Four Error later.

| | **K = 3** | **K = 5** | **K = 7** | **K = 9** | **K = 11** |
|---|---|---|---|---|---|
| **Euclidean** | 0.94272 | 0.95181 | 0.961 | 0.961 | 0.961 |
| **Manhattan** | 0.94181 | 0.9609 | 0.961 | 0.961 | 0.961 |
| **Chebyshev** | 0.95181 | 0.96181 | 0.97181 | 0.86 | 0.94272 |
| **Mahalanobis** | 0.89818 | 0.87 | 0.87 | 0.91667 | 0.87818 |
| **Minkowski** | 0.94272 | 0.95181 | 0.961 | 0.961 | 0.961 |

Table 3: Raw Data

For Z-Score normalized data, it is clear that when $\boldsymbol{K = 5}$ and the distance metric is ***Euclidean*** or ***Minkowski***, we got best accuracy. As you can see, raw data gave much more accuracy than z-score normalized data, but, in order to prove the result is correct or not, we will use Four Error later.

| | **K = 3** | **K = 5** | **K = 7** | **K = 9** | **K = 11** |
|---|---|---|---|---|---|
| **Euclidean** | 0.93272 | 0.96272 | 0.94363 | 0.94272 | 0.94272 |
| **Manhattan** | 0.94181 | 0.94272 | 0.94363 | 0.95181 | 0.95181 |
| **Chebyshev** | 0.92454 | 0.94454 | 0.93454 | 0.94363 | 0.94363 |
| **Mahalanobis** | 0.89818 | 0.87 | 0.87 | 0.86 | 0.87818 |
| **Minkowski** | 0.93272 | 0.96272 | 0.94363 | 0.94272 | 0.94272 |

Table 4: Z-Score Data

The results we have seen above is generated when the weights parameter is "uniform". To expand our experiment, we also tried when weight parameter is "distance". When we changed to "distance, for both raw and z-score normalized data, we got exactly the same accuracies. In order to validate our findings, we have used Four Error method.

```
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)

    trainDev_pred = model.predict(TrainDev_x)

    print("Train-Train Dev,   e1:", metrics.mean_squared_error(TrainDev_y, trainDev_pred),"\n")
    print("KNN Accuracy: ", 1 - metrics.mean_squared_error(TrainDev_y, trainDev_pred))

    dev_pred = model.predict(Dev_x)

    print("Train-Dev,   e2", metrics.mean_squared_error(Dev_y, dev_pred),"\n")
    print("KNN Accuracy: ", 1 - metrics.mean_squared_error(Dev_y, dev_pred))

    test_pred = model.predict(Test_x)

    print("Train-Test,   e3: ", metrics.mean_squared_error(Test_y, test_pred),"\n")
    print("KNN Accuracy: ", 1 - metrics.mean_squared_error(Test_y, test_pred))

    devTest_pred = model.predict(X_test)

    print("Train-(Dev+Test),   e4: ", metrics.mean_squared_error(Y_test, devTest_pred),"\n")
    print("KNN Accuracy: ", 1 - metrics.mean_squared_error(Y_test, devTest_pred))
```

Figure 12: Four Error Code

When we applied four error strategy, we got good accurracies but we did not get stable results. We even tried 2nd best and 3rd best, but they behaved same.
Our results:

```
knn = KNeighborsClassifier(n_neighbors=7, weights='uniform', metric="chebyshev")
fourError(X, Y, knn)
```

```
Train-Train Dev,   e1: 0.19047619047619047

KNN Accuracy:  0.8095238095238095
Train-Dev,   e2 0.045454545454545456

KNN Accuracy:  0.9545454545454546
Train-Test,   e3:  0.0

KNN Accuracy:  1.0
Train-(Dev+Test),   e4:  0.022222222222222223

KNN Accuracy:  0.9777777777777777
```

Figure 13: Four Error - Raw Data - K = 7

```
knn = KNeighborsClassifier(n_neighbors=5, weights='uniform', metric="minkowski")
fourError(X, Y, knn)
```

```
Train-Train Dev,   e1: 0.14285714285714285

KNN Accuracy:  0.8571428571428572
Train-Dev,   e2 0.0

KNN Accuracy:  1.0
Train-Test,   e3:  0.0

KNN Accuracy:  1.0
Train-(Dev+Test),   e4:  0.0

KNN Accuracy:  1.0
```

Figure 14: Four Error - Raw Data - K = 5 - Minkowski

```
knn = KNeighborsClassifier(n_neighbors=5, weights='uniform', metric="euclidean")
fourError(X, Y, knn)
```

```
Train-Train Dev,   e1: 0.14285714285714285

KNN Accuracy:  0.8571428571428572
Train-Dev,   e2 0.0

KNN Accuracy:  1.0
Train-Test,   e3:  0.0

KNN Accuracy:  1.0
Train-(Dev+Test),   e4:  0.0

KNN Accuracy:  1.0
```

Figure 15: Four Error - Raw Data - K = 5 - Euclidean

We called the classification report module to make sure our results are correct, and added to ***fourError*** method. Turns out, knn made a good job, but we could not get stable results.

```
Train-Train Dev,   e1: 0.09523809523809523

KNN Accuracy:  0.9047619047619048

Classification report

              precision    recall  f1-score   support

           0       1.00      1.00      1.00         7
           1       0.78      1.00      0.88         7
           2       1.00      0.71      0.83         7

    accuracy                           0.90        21
   macro avg       0.93      0.90      0.90        21
weighted avg       0.93      0.90      0.90        21

Train-Dev,   e2 0.0

KNN Accuracy:  1.0

Classification report

              precision    recall  f1-score   support

           0       1.00      1.00      1.00         7
           1       1.00      1.00      1.00         7
           2       1.00      1.00      1.00         8

    accuracy                           1.00        22
   macro avg       1.00      1.00      1.00        22
weighted avg       1.00      1.00      1.00        22

Train-Test,   e3:  0.0

KNN Accuracy:  1.0

Classification report

              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       1.00      1.00      1.00         8
           2       1.00      1.00      1.00         7

    accuracy                           1.00        23
   macro avg       1.00      1.00      1.00        23
weighted avg       1.00      1.00      1.00        23

Train-(Dev+Test),   e4:  0.0

KNN Accuracy:  1.0

Classification report

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       1.00      1.00      1.00        15
           2       1.00      1.00      1.00        15

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Figure 16: KNN Classification Report with four error

# 3 Naïve Bayes

It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. It is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Figure 17: Bayes' Theorem

## 3.1 Classifiers

### 3.1.1 Gaussian Naive Bayes

In Gaussian Naive Bayes, continuous values associated with each feature are assumed to be distributed according to a Gaussian distribution. A Gaussian distribution is also called Normal distribution.

### 3.1.2 Multinomial Naive Bayes

In Multinomial Naive Bayes, feature vectors represent the frequencies with which certain events have been generated by a multinomial distribution. This is the event model typically used for document classification.

### 3.1.3 Complement Naive Bayes

Complement Naive Bayes is an adaptation of the standard Multinomial Naive Bayes algorithm that is particularly suited for imbalanced data sets. Specifically, Complement Naive Bayes uses statistics from the complement of each class to compute the model's weights.

### 3.1.4 Bernoulli Naive Bayes

Bernoulli Naive Bayes used for data that is distributed according to multivariate Bernoulli distributions. There may be multiple features but each one is assumed to be a binary-valued(boolean) variable. Therefore, it requires samples to be represented as binary-valued feature vectors.

### 3.1.5 Categorical Naive Bayes

Categorical Naive Bayes used for categorically distributed data. It assumes that each feature, which is described by the index , has its own categorical distribution.

## 3.2 Experiments

In order to find best Naive Bayes predictor, we tried all Naive Bayes classifiers which are gaussian, multinomial, complement, bernoulli and categorical naive bayes. Then, we decided our final model.

### 3.2.1 Gaussian Naive Bayes

In scikit learn, Gaussian Naive Bayes classifier, *GaussianNB*, takes only two parameters which we decided to use as default. Therefore, in order to find best model, we tried different percentages for splitting our dataset into training and testing sets, and different random state numbers. By looking at the accuracies, we decided best split strategy .

| | **%20 - %80** | **%25 - %75** | **%30 - %70** |
|---|---|---|---|
| **5-Fold** | 93.3333 | 93.7549 | 93.3333 |
| **10-Fold** | 94.1666 | 94.6969 | 93.3636 |
| **Random One Holdout** | 91.6666 | 95.6521 | 85.7142 |
| **Stratified One Holdout** | 100.0 | 95.6521 | 90.4761 |
| **Unseen Data(Test Set)** | 96.6666 | 97.3684 | 97.7777 |

Table 5: Accuracies for different portions of train and test sets

We used 12 as random state number for the table 5 since it gave more stable results. As you can see, accuracies are mostly close to each other. Therefore we chose %30-%70, since it has the best accuracy. However, we also need to make sure that our model is stable. In order to decide that, we checked four errors.

| **Train-Train Dev e1** | **Train-Dev e2** | **Train-Test e3** | **Train-(Dev+Test) e4** |
|---|---|---|---|
| 0.0476 | 0.0454 | 0.0434 | 0.0444 |

Table 6: Four errors for Gaussian Naive Bayes predictor

Four errors in table 6 shows that our model is stable if use 12 as random state number for data split.

### 3.2.2 Multinomial Naive Bayes

In scikit learn, Multinomial Naive Bayes classifier, *MultinomialNB*, has a 'fit_prior' parameter which decides whether model will learn class prior probabilities or not. If fit_prior = false, it uses a uniform prior. We tried both of them to decide the best predictor using different validation techniques. We also tested our model with the unseen data.

| | **fit_prior = False** | **fit_prior = True** |
|---|---|---|
| **5-Fold** | 93.7944 | 88.4980 |
| **10-Fold** | 93.8636 | 86.6666 |
| **Random One Holdout** | 91.3043 | 95.6521 |
| **Stratified One Holdout** | 86.9565 | 86.9565 |
| **Unseen Data(Test Set)** | 88.9681 | 86.8421 |

Table 7: Accuracies for different fit_prior parametrs and validation techniques

For the experiment in table 7, we used %25-%75 portions of data for testing and training sets. We also used used 9 for the random state number during split operation. Otherwise, we did not get good results as above. When we look at the accuracies in table 7, we can say that fit_prior parameter does not affect our model so much. Therefore, we decided to choose fit_prior parameter as 'False'. In order to make sure that we make a good decision, we checked four errors.

| **Train-Train Dev e1** | **Train-Dev e2** | **Train-Test e3** | **Train-(Dev+Test) e4** |
|---|---|---|---|
| 0.0476 | 0.0454 | 0.0434 | 0.0444 |

Table 8: Four errors for Multinomial Naive Bayes predictor

Again we used 9 for the random state number during split in table 8. You can see that, our model is quite stable.

### 3.2.3 Complement Naive Bayes

In this experiment, first, we decided how to split our data and observe accuracies for different validation techniques and test result with the unseen data. In all experiments, random state number is 9.

| | %20 - %80 | %25 - %75 | %30 - %70 |
|---|---|---|---|
| **5-Fold** | 65.8333 | 66.9960 | 65.7142 |
| **10-Fold** | 65.8333 | 66.9696 | 65.8181 |
| **Random One Holdout** | 87.5 | 73.9130 | 66.6666 |
| **Stratified One Holdout** | 66.6666 | 65.2173 | 66.6666 |
| **Unseen Data(Test Set)** | 70.0 | 65.7894 | 68.8888 |

Table 9: Accuracies for different portions of train and test sets

The table 9 shows that &30-%70 gives more reliable results for random state number 9. Therefore, we continued with this split strategy, and tried different parameters.

| | fit_prior = False | fit_prior = True |
|---|---|---|
| **norm = False** | 65.8181 | 65.8181 |
| **norm = True** | 30.6363 | 30.63636 |

Table 10: Accuracies for different fit_prior and norm values

According to the table 10, we decided to use fit_prior = False and norm = True which are default values in complement naive bayes classifier. As you can see, result are not good at all. The reason behind this is that complement naive bayes uses statistics from the complement of each class to compute the model's weights. However, in our dataset, most of the samples' feature values are close to each other. There is no significant difference to use complement of each class. Finally, we also checked four errors and looked at the stability of our model.

| Train-Train Dev e1 | Train-Dev e2 | Train-Test e3 | Train-(Dev+Test) e4 |
|---|---|---|---|
| 0.3333 | 0.3636 | 0.3043 | 0.3333 |

Table 11: Four errors for Complement Naive Bayes predictor

The table 11 shows that, although our model gives bad accuracies, it is quite stable.

### 3.2.4 Bernoulli Naive Bayes

In scikit learn, Bernoulli Naive Bayes classifier, *BernoulliNB*, assumes that data is already consist of binary vectors unless you change the 'binarize' parameter. First of all, qithout doing any changes, we tried different split strategies with different validation techniques.

| | %20 - %80 | %25 - %75 | %30 - %70 |
|---|---|---|---|
| **5-Fold** | 33.3333 | 35.7312 | 33.3333 |
| **10-Fold** | 33.3333 | 35.7575 | 33.1818 |
| **Random One Holdout** | 12.5 | 30.4347 | 23.8095 |
| **Stratified One Holdout** | 33.3333 | 34.7826 | 33.3333 |

Table 12: Accuracies for different portions of train and test sets

You can see in table 12 that bernoulli naive bayes classifier fails since our dataset do not have binary/boolean features. However, we decided to continue with %25-%75 data split strategy. We also used 5-Fold to do validation during tuning our model.
Next step was using 'binarize' parameter and trying to map our data to booleans.

| 0 | 0.25 | 0.5 | 0.75 | 1 | 1.25 | 1.5 | 1.75 | 2 | 2.25 | 2.5 |
|---|---|---|---|---|---|---|---|---|---|---|
| 35.73 | 57.15 | 66.99 | 66.99 | 66.99 | 66.99 | 85.73 | 92.92 | 80.35 | 73.12 | 66.99 |

Table 13: Accuracies for different binarize thresholds

When we used binarize threshold as 1.75, we got the best accuracy. However, the table below shows that is not stable.

| Train-Train Dev e1 | Train-Dev e2 | Train-Test e3 | Train-(Dev+Test) e4 |
|---|---|---|---|
| 0.0476 | 0.1363 | 0.0434 | 0.0888 |

Table 14: Four errors for binarize = 1.75

We also tried binarize threshold as 1.5. We got four errors below. It is more stable than binarize= 1.75, however, still we cannot say model is stable. Final accuracy was 86.85.

| Train-Train Dev e1 | Train-Dev e2 | Train-Test e3 | Train-(Dev+Test) e4 |
|---|---|---|---|
| 0.0952 | 0.0909 | 0.1304 | 0.1111 |

Table 15: Four errors for binarize = 1.5

### 3.2.5 Categorical Naive Bayes

First, we decided to observe our data since this classifiers assumes that data is categorical. We saw that feature values in Iris dataset are repeated most of the time. Therefore, we can think them like categories. To make sure, we also checked outliers using functions in table 18.

```python
def displayAccuracy(X, Y):

    NaiveBayes(X, Y)
    #DecisionTree(X, Y)
```

```python
def zValues(df):

    cols = list(df.columns)
    cols.remove('Index')

    for col in cols:
        col_zscore = col + '_zscore'
        df[col_zscore] = (df[col] - df[col].mean())/df[col].std(ddof=0)

    return df
```

```python
def outliers(df):
    return df.loc[(df._1_zscore > 2.5) | (df._2_zscore > 2.5) | (df._2_zscore > 2.5) | (df._3_zscore > 2.
5)]
```

```python
def subDatasets(df):
    target0 = []
    target1 = []
    target2 = []
    for row in df.itertuples():
        if row.target == 0:
            target0.append(row)
            target0_df = pd.DataFrame(target0)
        elif row.target == 1:
            target1.append(row)
            target1_df = pd.DataFrame(target1)
        else:
            target2.append(row)
            target2_df = pd.DataFrame(target2)

    dfs = [target0_df, target1_df, target2_df]

    for df in dfs:
        df.drop(columns=['target'])

    return target0_df, target1_df, target2_df
```

Figure 18: Functions for finding outliers for each class

We found that, class-1 have three outliers and class-3 have only one outlier where class-2 have no outlier at all. Therefore, since number of outliers quite a few, we did not remove them from the dataset, and we thought data is categorical in our experiments. we decided our data split strategy with different validation techniques. Then, we tuned parameters and tried to get the best accuracy.

| | %20 - %80 | %25 - %75 | %30 - %70 |
|---|---|---|---|
| **5-Fold** | 91.6666 | 92.8063 | 91.4285 |
| **10-Fold** | 91.6666 | 92.7272 | 91.2727 |
| **Random One Holdout** | 83.3333 | 100.0 | 90.4761 |
| **Stratified One Holdout** | 91.6666 | 95.6521 | 100.0 |
| **Unseen Data(Test Set)** | 100.0 | 94.7368 | 95.5555 |

Table 16: Accuracies for different portions of train and test sets

In table 16, we can see that %25-%75 gave more stable result. Therefore, continued with this portion of training set. We also used fit_prior parameter as default since it did not change accuracy in our case.

| Train-Train Dev e1 | Train-Dev e2 | Train-Test e3 | Train-(Dev+Test) e4 |
|---|---|---|---|
| 0.0 | 0.0454 | 0.0434 | 0.0444 |

Table 17: Four errors for Categorical Naive Bayes predictor

When we look at the table 17, we can say that our model is kind of stable. The reason behind this accuracies is, our feature values

# 4 Decision Tree

The decision tree is a largely used non-parametric effective machine learning modeling technique for regression and classification problems. In order to find solutions, a decision tree makes sequential, hierarchical decision about the outcomes variable based on the predictor data.
In our experiments, we will try different validation techniques, split strategies against different split metrics (i.e, entropy and Gini), max_depth values and class weights. Then, based on our experiments, we will decide best classification strategy for Iris dataset using the decision tree classifier.

## 4.1 Split Metrics

### 4.1.1 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 19: Gini Calculation

### 4.1.2 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 20: Entropy Calculation

## 4.2 Experiments

First of all, we divided our data as 70% for training and 30% for testing. We tried 5-Fold, 10-Fold, Random One Holdout and Stratified One Hold Out with different depth values. As you can see in figure:21, after deciding best validation technique and depth value, we tried to find best split strategy. Then, using depth value and split strategies we have decided, we tried different class weights, and decided our final parameters. Finally, we tested our model with unseen data, and checked stability of the model using four errors.

```
def DecisionTree(X, Y):

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

    # 5-Fold
    print("\n5-Fold: ")
    tuningDepth(X_train, Y_train, 0)

    # 10-Fold
    print("\n10-Fold: ")
    tuningDepth(X_train, Y_train, 1)

    # Random One Holdout
    print("\nRandom One Holdout: ")
    tuningDepth(X_train, Y_train, 2)

    # Stratified One Holdout
    print("\nStratified One Holdout: ")
    tuningDepth(X_train, Y_train, 3)

    #
    #   Continue with 10-Fold, Depth = 3
    #

    print("5-Fold, Depth=5\n")
    tuningSplit(X_train, Y_train)

    #
    #   Continue with criterion = 'gini', splitter = 'best', min_samples_split = 2
    #                   all of them are default values
    print("5-Fold, depth = 5, criterion = 'gini', splitter = 'best, min_samples_split = 2\n")
    tuningClassWeight(X_train, Y_train)

    #
    #   Continue with class_weight = None,   default
    #

    clf = DecisionTreeClassifier(max_depth = 3, random_state = 0)
    clf.fit(X_train, Y_train)

    y_pred = clf.predict(X_test)
    print("Accuracy: ", metrics.accuracy_score(Y_test, y_pred)*100)
```

Figure 21: Decision Tree Predictor Function

For tuning max_depth value, we have the function in figure:22 which takes validation technique and data. It creates model for all max_depth values 1 to 7 and none depth value which means nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

```
def tuningDepth(X_train, Y_train, val):

    max_depth_range = list(range(1, 10))

    for depth in max_depth_range:
        if (val == 0):
            clf = DecisionTreeClassifier(max_depth = depth, random_state = 0)
            clf.fit(X_train, Y_train)

            cv = cross_val_score(clf, X_train, Y_train, cv=5, scoring='accuracy')
            print("Depth: ", depth, " Accuracy: ",cv.mean()*100)

        elif (val == 1):
            clf = DecisionTreeClassifier(max_depth = depth, random_state = 0)
            clf.fit(X_train, Y_train)

            cv = cross_val_score(clf, X_train, Y_train, cv=10, scoring='accuracy')
            print("Depth: ", depth, " Accuracy: ",cv.mean()*100)

        elif(val == 2):
            x_train, x_test, y_train, y_test = randomOneHoldout(X_train, Y_train)

            clf = DecisionTreeClassifier(max_depth = depth, random_state = 0)
            clf.fit(x_train, y_train)

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

        elif(val == 3):
            x_train, x_test, y_train, y_test = stratifiedOneHoldout(X_train, Y_train)

            clf = DecisionTreeClassifier(max_depth = depth, random_state = 0)
            clf.fit(x_train, y_train)

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

        else:
            print("Invalid validation tech.")
```

Figure 22: Tuning max_depth parameter in decision tree

| | None | d = 1 | d = 2 | d = 3 | d = 4 | d = 5 | d = 6 | d = 7 |
|---|---|---|---|---|---|---|---|---|
| **5-Fold** | 94.2857 | 69.5238 | 93.3333 | 94.2857 | 93.3333 | 93.3333 | 94.2857 | 94.2857 |
| **10-Fold** | 96.0909 | 69.5454 | 95.1818 | 98.0 | 96.0909 | 96.0909 | 96.0909 | 96.0909 |
| **Random One Holdout** | 100.0 | 42.857 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| **Stratified One Holdout** | 100.0 | 71.4285 | 95.2380 | 95.2380 | 90.4761 | 90.4761 | 90.4761 | 90.4761 |

Table 18: Accuracies for different max_depth(d) values and validation techniques

As you can see above, when there is no max_depth value, random one holdout and stratified one holdout validation techniques gave 100.0 accuracy. However, when we tested the models with unseen data, accuracies are dropped. We can say that

there is over-fitting and these techniques are not reliable for none max_dept value. Moreover, random one holdout always gives 100.0 for this data(70% training, 30% testing). The reason why we chose this distribution although there is an over-fitting problem with random one holdout is that we got best result which is 98.0% with 10-fold cross validation. The table also shows that when we increase max_depth, accuracy reaches maximum at some point, then it remains same. For our experiment, we can say that we should use 10-fold for validation, and 3 as max_depth value.

Our next step for finding the best decision tree predictor is to decide splitting strategies. We used function in figure:23 to compare split criterions(gini or entropy) and min_samples_split parameter.

```
def tuningSplit(X_train, Y_train):
    criterion = ["gini", "entropy"]
    splitter = ["best", "random"]

    for i in criterion:
        for j in splitter:
            clf = DecisionTreeClassifier(criterion = i, splitter = j, max_depth = 3, random_state = 0)
            clf.fit(X_train, Y_train)

            cv = cross_val_score(clf, X_train, Y_train, cv=10, scoring='accuracy')
            print("Criterion: ", i, "   Splitter: ", j, "   Accuracy: ", cv.mean()*100)

    for i in range(2, 10):
        clf = DecisionTreeClassifier(max_depth = 3, min_samples_split = i, random_state = 0)
        clf.fit(X_train, Y_train)

        cv = cross_val_score(clf, X_train, Y_train, cv=10, scoring='accuracy')
        print("min_samples_split: ", i, "   Accuracy: ", cv.mean()*100)
```

Figure 23: Tuning split strategies in decision tree

| | **criterion = 'gini'** | **criterion = 'entropy'** |
|---|---|---|
| **splitter = 'best'** | 98.00 | 97.09 |
| **splitter = 'random'** | 95.18 | 95.18 |

Table 19: Accuracies for different split techniques

| **min_samples_split** | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| **Accuracy** | 98.0 | 98.0 | 98.0 | 98.0 | 98.0 |

Table 20: Accuracies for different min_samples_split

Table 19 shows that choosing the best split gives better result than choosing the best random split. Moreover, looking Gini impurity rather than entropy gave high accuracy. We also observed that min_samples_split parameter has no effect on our model(20). Accuracy remained 98.0%. Therefore, we decided to continue with 'Gini', 'best splitter' and 'min_samples_split = 2' as a split strategy.

| class_weight = None | class_weight = 'balanced' |
|---|---|
| 98.0 | 98.0 |

Table 21: Accuracies for no class weight and balanced class weight

Finally, we tried different class weights. As you can see in table 21, we created our model with no class weight and balanced weight. Both of them are gave same accuracy. That means, importance of the features in our dataset are same or very close to each other.
As a result, we created a decision tree predictor with max_depth = 3, criterion = 'gini', splitter = 'best', min_samples_split = 2 and class_weight = None.
To make sure whether our final model is stable or not, we split our data into four, and checked four errors. As you can see in table 22, all errors are around 0.04. Therefore, we can say that our decision tree predictor is stable.

| Train-Train Dev e1 | Train-Dev e2 | Train-Test e3 | Train-(Dev+Test) e4 |
|---|---|---|---|
| 0.0476 | 0.0454 | 0.0434 | 0.0444 |

Table 22: Four errors for the decision tree predictor

# 5 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.

## 5.1 Experiments

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

```
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 24: SVM code

We generally look our SVM model's result, and we decided to move on with 5 or 10 Fold because other validation techniques gave zero error percentage, and we taught that there is over-fitting problem. After we decided to go with 5 Fold, we started to feed our method with Kernel parameters. We used four different kernel parameters which are Linear, Poly, Rbf, Sigmoid.

```
Raw:
Kernel: linear
5 Fold
SVM Accuracy:  0.9714285714285715
Kernel: poly
5 Fold
SVM Accuracy:  0.9619047619047618
Kernel: rbf
5 Fold
SVM Accuracy:  0.9714285714285713
Kernel: sigmoid
5 Fold
SVM Accuracy:  0.3714285714285714
```

Figure 25: SVM Kernel Result

As we can see above, we got the worst result by Sigmoid, and we got the best result by Linear and Rbf. We could continue with Linear  Rbf. Then, we continued with class weight. In class_weight parameter, we have two option which are None and Balanced. We will try use with Rgf and Linear kernel

```
Kernel: linear
5 Fold
SVM Accuracy:  0.9714285714285715
Kernel: linear
5 Fold
SVM Accuracy:  0.980952380952381
```

Figure 26: SVM Linear kernel result

```
Kernel: rbf
5 Fold
SVM Accuracy:  0.9714285714285713
Kernel: rbf
5 Fold
SVM Accuracy:  0.9714285714285713
```

Figure 27: SVM Linear kernel result

As we can see from the results, we increased our accuracy on Linear kernel, but on rbf, we could not increase or decrease the accuracy. Finally, we made changes on gamma value. During our experiment, we figured out that Linear kernel only works with when gamma value "auto".

```
SVM Accuracy:  0.9714285714285715
Kernel: linear - Weight: None - Gamma: auto
5 Fold
SVM Accuracy:  0.9714285714285715
Kernel: linear - Weight: balanced - Gamma: auto
```

Figure 28: SVM Linear kernel result

```
5 Fold
SVM Accuracy:  0.980952380952381
Kernel: rbf - Weight: None - Gamma: auto
5 Fold
SVM Accuracy:  0.9714285714285713
Kernel: rbf - Weight: None - Gamma: scale
5 Fold
SVM Accuracy:  0.9523809523809523
Kernel: rbf - Weight: balanced - Gamma: auto
5 Fold
SVM Accuracy:  0.9714285714285713
Kernel: rbf - Weight: balanced - Gamma: scale
5 Fold
SVM Accuracy:  0.9619047619047618
```

Figure 29: SVM Linear kernel result

As a result, we got quite good results, but I order to validate it, we need to look four error result.

```
Train-Train Dev,   e1: 0.09523809523809523

KNN Accuracy:  0.9047619047619048
Train-Dev,   e2 0.0

KNN Accuracy:  1.0
Train-Test,   e3:  0.0

KNN Accuracy:  1.0
Train-(Dev+Test),   e4:  0.0

KNN Accuracy:  1.0
```

Figure 30: SVM Four Error result

We got really good results, but we suspect that there could be problem with our models since we got 100% matching. Therefore, we decided to look at precision, recall and f1-score using built-in method in Sklearn.

```
Train-Train Dev,   e1: 0.19047619047619047

KNN Accuracy:  0.8095238095238095

Classification report

              precision    recall  f1-score   support

           0       1.00      1.00      1.00         7
           1       0.71      0.71      0.71         7
           2       0.71      0.71      0.71         7

    accuracy                           0.81        21
   macro avg       0.81      0.81      0.81        21
weighted avg       0.81      0.81      0.81        21

Train-Dev,   e2 0.045454545454545456

KNN Accuracy:  0.9545454545454546

Classification report

              precision    recall  f1-score   support

           0       1.00      1.00      1.00         7
           1       0.88      1.00      0.93         7
           2       1.00      0.88      0.93         8

    accuracy                           0.95        22
   macro avg       0.96      0.96      0.96        22
weighted avg       0.96      0.95      0.95        22

Train-Test,   e3:  0.0

KNN Accuracy:  1.0

Classification report

              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       1.00      1.00      1.00         8
           2       1.00      1.00      1.00         7

    accuracy                           1.00        23
   macro avg       1.00      1.00      1.00        23
weighted avg       1.00      1.00      1.00        23

Train-(Dev+Test),   e4:  0.022222222222222223

KNN Accuracy:  0.9777777777777777

Classification report

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       0.94      1.00      0.97        15
           2       1.00      0.93      0.97        15

    accuracy                           0.98        45
   macro avg       0.98      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
```

Figure 31: SVM Classification Report with four error

After looking at the classification report, we can safely say that we got correct result. As a result, most suitable Kernel can boost our accuracy, and class_weight and gamma value did not do much improvement with this model on iris data-set.

# 6 Final Results

| | **Accuracy** | **Stability** |
|---|---|---|
| **K-Nearest Neighbors** | 97.181 | False |
| **Naive Bayes (Gaussian)** | 97.777 | True |
| **Decision Tree** | 98.0 | True |
| **Support Vector Machine** | 98.095 | False |

Table 23: Final Results for All Classifiers

# 7 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.

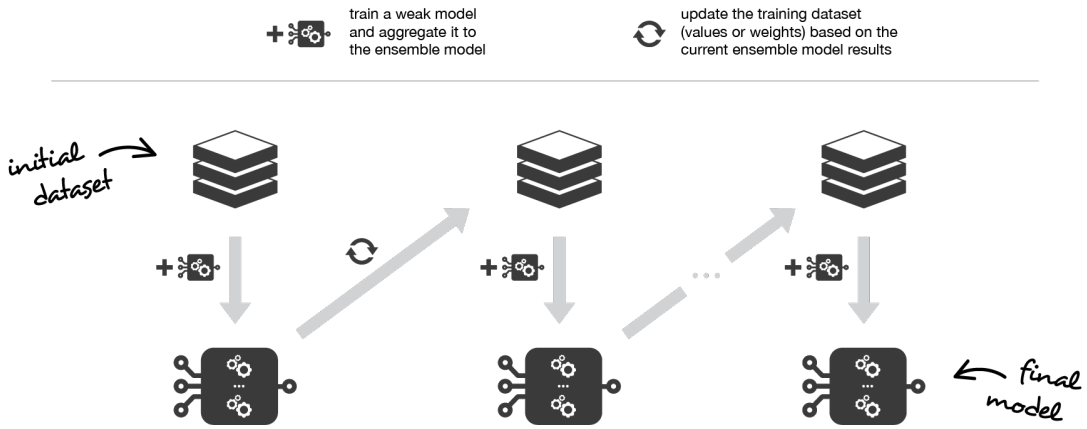


Figure 32: Boosting

## 7.1 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, but if the model does not have weight like knn model, we cannot boost the method.

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

Figure 33: Adaboost Formula

## 7.2 Gradient Boosting

In gradient boosting, the ensemble model we try to build is also a weighted sum of weak learners. It has the same formula with adaboost, but the working principle is quite different. In sklearn, unlike adaboost, we cannot feed models that we created as a parameter to gradient boost method. It works by itself.

## 7.3 Experiment

For the experiment, we chose ***Complement Naive Bayes*** as worst model and ***Decision Tree*** as best model by comparing our results. We will boost these both model using adaboost and gradient boost. Since we cannot use our models on gradient 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 34: Adaboost and Gradient Boost

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=9)
#Worst model
complement = ComplementNB()
learning_rate = [0.0001, 0.001, 0.01, 0.1, 1 ,2 ,3 ,4, 5]

result_ada_1 = []
result_gradient_1 = []
for i in learning_rate: #i -> Learning Rate
    x = []
    y = []
    for j in range(50, 150): #j -> N estimators
        x.append(AdaBoost(complement, j, i, X_train, Y_train, X_test, Y_test))
        y.append(GradientBoost(j, i, X_train, Y_train, X_test, Y_test))
    result_ada_1.append(x)
    result_gradient_1.append(y)
```

Figure 35: Boosting Worst Model

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=0)
#best model
clf = DecisionTreeClassifier(max_depth = 3)
learning_rate = [0.0001, 0.001, 0.01, 0.1, 1 ,2 ,3 ,4, 5]

result_ada_2 = []
result_gradient_2 = []
for i in learning_rate: #i -> Learning Rate
    x = []
    y = []
    for j in range(50, 150): #j -> N estimators
        x.append(AdaBoost(clf, j, i, X_train, Y_train, X_test, Y_test))
        y.append(GradientBoost(j, i, X_train, Y_train, X_test, Y_test))
    result_ada_2.append(x)
    result_gradient_2.append(y)
```

Figure 36: Boosting Best Model

## 7.4 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

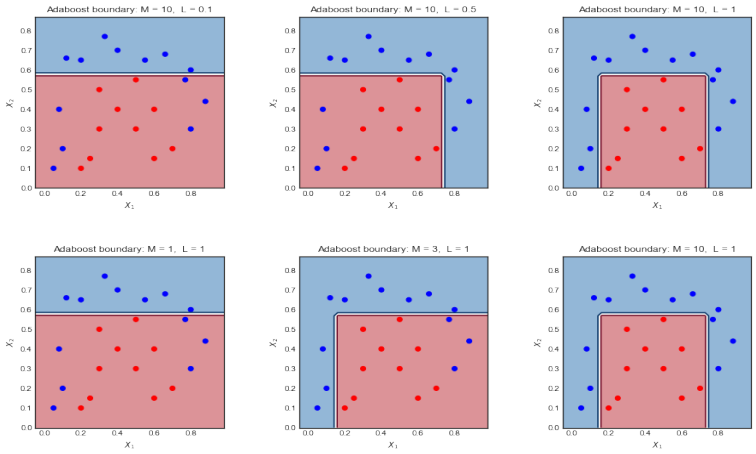


Figure 37: Comparison of Learning Rate and Estimators

By looking at our result, gradient boost gave the most accurate and highest accuracy.

# 8 Appendix

## 8.1 Project Link

https://github.com/nisanuro/CNG562-Assignment-2

## 8.2 Code

```
# -*- coding: utf-8 -*-
"""CNG562-Assignment2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/github/nisanuro/CNG562-Assignment-2/blob/master/CNG562_Assignment2.ipynb
"""

# Commented out IPython magic to ensure Python compatibility.
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, KFold, StratifiedKFold, cross_val_score
from mpl_toolkits.mplot3d import Axes3D
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics, datasets, preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.svm import LinearSVC, SVC

from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from sklearn.metrics import classification_report
from sklearn.naive_bayes import GaussianNB, BernoulliNB, CategoricalNB, ComplementNB, MultinomialNB
from sklearn.tree import DecisionTreeClassifier

# %matplotlib inline

def dataVisualizaion(iris):
    x_index = 0
    y_index = 1

    formatter = plt.FuncFormatter(lambda i, *args: iris.target_names[int(i)])
```

```
34      plt.figure(figsize=(5, 4))
        plt.scatter(iris.data[:, x_index], iris.data[:, y_index], c=
                                        iris.target)
36      plt.colorbar(ticks=[0, 1, 2], format=formatter)
        plt.xlabel(iris.feature_names[x_index])
38      plt.ylabel(iris.feature_names[y_index])

40      plt.tight_layout()
        plt.show()
42
    def threeDVisualization(X, y):
44
        scaler = StandardScaler()
46      X_scaled = scaler.fit_transform(X)

48      fig = plt.figure(1, figsize=(16, 9))
        ax = Axes3D(fig, elev=-150, azim=110)
50      X_reduced = PCA(n_components=3).fit_transform(X_scaled)
        ax.scatter(X_reduced[:, 0], X_reduced[:, 1], X_reduced[:, 2], c
                                        =y, cmap=plt.cm.Set1, edgecolor='
                                        k', s=40)
52      ax.set_title("First three PCA directions")
        ax.set_xlabel("1st eigenvector")
54      ax.w_xaxis.set_ticklabels([])
        ax.set_ylabel("2nd eigenvector")
56      ax.w_yaxis.set_ticklabels([])
        ax.set_zlabel("3rd eigenvector")
58      ax.w_zaxis.set_ticklabels([])

60      plt.show()
        print("The number of features in the new subspace is ",
                                        X_reduced.shape[1])
62
        return X_reduced
64
    def randomOneHoldout(X_train, Y_train):
66      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
68
    def stratifiedOneHoldout(X_train, Y_train):
70      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
72
    def NaiveBayes(X, Y):
74      X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                        test_size=0.25, random_state=9)
```

```
76      # Gaussian Naive Bayes
        gaussian = GaussianNB()
78      NaiveBayesValidations(X_train, Y_train, gaussian)

80      y_pred = gaussian.predict(X_test)
        print("Accuracy - unseen data: ", metrics.accuracy_score(Y_test
                                    , y_pred)*100)
82
        fourError(X, Y, gaussian)
84

86      # Multinomial Naive Bayes
        multinomial = MultinomialNB(fit_prior=True)
88      NaiveBayesValidations(X_train, Y_train, multinomial)

90      y_pred = multinomial.predict(X_test)
        print("Accuracy - unseen data: ", metrics.accuracy_score(Y_test
                                    , y_pred)*100)
92
        fourError(X, Y, multinomial)
94

96      # Bernoulli Naive Bayes

98      bernoulli = BernoulliNB(binarize = 1.75)
        bernoulli.fit(X_train, Y_train)
100     #NaiveBayesValidations(X_train, Y_train, bernoulli)

102
        y_pred = bernoulli.predict(X_test)
104     print("Accuracy - unseen data: ", metrics.accuracy_score(Y_test
                                    , y_pred)*100)

106     fourError(X, Y, bernoulli)

108     #BernoulliBinarize(X_train, Y_train)

110
        # Complement Naive Bayes
112
        complement = ComplementNB()
114     NaiveBayesValidations(X_train, Y_train, complement)

116     y_pred = complement.predict(X_test)
        print("Accuracy - unseen data: ", metrics.accuracy_score(Y_test
                                    , y_pred)*100)
118
        fourError(X, Y, complement)
120
```

```
    #complementTuning(X_train, Y_train)
122
    # Categorical Naive Bayes
124 categorical = CategoricalNB(fit_prior = False)
    NaiveBayesValidations(X_train, Y_train, categorical)
126
def complementTuning(X_train, Y_train):
128     fit_prior = [True, False]
    norm = [True, False]
130
    for i in fit_prior:
132         for j in norm:
                model = ComplementNB(fit_prior = i, norm = j)
134
                cv = cross_val_score(model, X_train, Y_train, cv=10,
                                     scoring='accuracy')
136
                print("fit_prior = ", i, "   norm = ", j, "   Accuracy:
                                      ", cv.mean()*100)
138
def BernoulliBinarize(X_train, Y_train):
140     binarize = [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 2.25, 2.
                                    5, 2.75, 3]

142     for i in binarize:

144         model = BernoulliNB(binarize = i)
            cv = cross_val_score(model, X_train, Y_train, cv=5, scoring
                                    ='accuracy')
146         print("Binarize = ", i, "    Accuracy: ", cv.mean()*100)

148 def NaiveBayesValidations(X_train, Y_train, model):
    print("\n" + str(model).split('N')[0] + " Naive Bayes Accuracy\
                                    n")
150     # 5-Fold
    cv = cross_val_score(model, X_train, Y_train, cv=5, scoring='
                                    accuracy')
152
    print("5-Fold: ", cv.mean()*100)
154
    # 10-Fold
156 cv = cross_val_score(model, X_train, Y_train, cv=10, scoring='
                                    accuracy')

158 print("10-Fold: ", cv.mean()*100)

160 # Random One Holdout
    x_train, x_test, y_train, y_test = randomOneHoldout(X_train,
                                    Y_train)
162
```

```
    model.fit(x_train, y_train)
    y_pred = model.predict(x_test)

    print("Random One Holdout: ", metrics.accuracy_score(y_test,
                                    y_pred)*100)

    # Stratified One Holdout
    x_train, x_test, y_train, y_test = stratifiedOneHoldout(X_train
                                    , Y_train)

    model.fit(x_train, y_train)
    y_pred = model.predict(x_test)

    print("Stratified One Holdout: ", metrics.accuracy_score(y_test
                                    , y_pred)*100)

def DecisionTree(X, Y):

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

    # 5-Fold
    print("\n5-Fold: ")
    tuningDepth(X_train, Y_train, 0)

    # 10-Fold
    print("\n10-Fold: ")
    tuningDepth(X_train, Y_train, 1)

    # Random One Holdout
    print("\nRandom One Holdout: ")
    tuningDepth(X_train, Y_train, 2)

    # Stratified One Holdout
    print("\nStratified One Holdout: ")
    tuningDepth(X_train, Y_train, 3)

    #
    #   Continue with 10-Fold, Depth = 3
    #

    print("5-Fold, Depth=5\n")
    tuningSplit(X_train, Y_train)

    #
    #   Continue with criterion = 'gini', splitter = 'best',
                                    min_samples_split = 2
    #                       all of them are default values
    print("5-Fold, depth = 5, criterion = 'gini', splitter = 'best,
                                     min_samples_split = 2\n")
```

```
    tuningClassWeight(X_train, Y_train)

    #
    #   Continue with class_weight = None,   default
    #

    clf = DecisionTreeClassifier(max_depth = 3)
    '''
    clf.fit(X_train, Y_train)

    y_pred = clf.predict(X_test)
    print("Accuracy: ", metrics.accuracy_score(Y_test, y_pred)*100)
    '''
    fourError(X, Y, clf)

def tuningClassWeight(X_train, Y_train):
    # No class weight
    clf = DecisionTreeClassifier(max_depth = 3, random_state = 0)
    clf.fit(X_train, Y_train)

    cv = cross_val_score(clf, X_train, Y_train, cv=10, scoring='
                                accuracy')
    print("Class weight: None           Accuracy: ", cv.mean()*100)

    # Balanced class weight
    clf = DecisionTreeClassifier(max_depth = 3, random_state = 0,
                                class_weight ='balanced')
    clf.fit(X_train, Y_train)

    cv = cross_val_score(clf, X_train, Y_train, cv=10, scoring='
                                accuracy')
    print("Class weight: Balanced       Accuracy: ", cv.mean()*100)

def tuningSplit(X_train, Y_train):
    criterion = ["gini", "entropy"]
    splitter = ["best", "random"]

    for i in criterion:
        for j in splitter:
            clf = DecisionTreeClassifier(criterion = i, splitter =
                                j, max_depth = 3, random_state =
                                0)
            clf.fit(X_train, Y_train)

            cv = cross_val_score(clf, X_train, Y_train, cv=10,
                                scoring='accuracy')
            print("Criterion: ", i, "   Splitter: ", j, "
                                Accuracy: ", cv.mean()*100)

    for i in range(2, 10):
```

```
250         clf = DecisionTreeClassifier(max_depth = 3,
                                         min_samples_split = i,
                                         random_state = 0)
            clf.fit(X_train, Y_train)
252
            cv = cross_val_score(clf, X_train, Y_train, cv=10, scoring=
                                         'accuracy')
254         print("min_samples_split: ", i, "    Accuracy: ", cv.mean()*
                                         100)

256 def tuningDepth(X_train, Y_train, val):

258     max_depth_range = list(range(1, 10))
        max_depth_range.append(str("None"))
260


262     for depth in max_depth_range:
            if (val == 0):
264             if(depth == "None"):
                    clf = DecisionTreeClassifier(random_state = 0)
266                 clf.fit(X_train, Y_train)
                else:
268                 clf = DecisionTreeClassifier(max_depth = depth,
                                         random_state = 0)
                    clf.fit(X_train, Y_train)
270
                cv = cross_val_score(clf, X_train, Y_train, cv=5,
                                         scoring='accuracy')
272             accuracy = cv.mean()*100
                print("Depth: ", depth, " Accuracy: ", accuracy)
274
            elif (val == 1):
276             if(depth == "None"):
                    clf = DecisionTreeClassifier(random_state = 0)
278                 clf.fit(X_train, Y_train)
                else:
280                 clf = DecisionTreeClassifier(max_depth = depth,
                                         random_state = 0)
                    clf.fit(X_train, Y_train)
282
                cv = cross_val_score(clf, X_train, Y_train, cv=10,
                                         scoring='accuracy')
284             accuracy = cv.mean()*100
                print("Depth: ", depth, " Accuracy: ",accuracy)
286
            elif(val == 2):
288             x_train, x_test, y_train, y_test = randomOneHoldout(
                                         X_train, Y_train)

290             if(depth == "None"):
```

```
                clf = DecisionTreeClassifier(random_state = 0)
292             clf.fit(X_train, Y_train)
            else:
294             clf = DecisionTreeClassifier(max_depth = depth,
                                 random_state = 0)
                clf.fit(x_train, y_train)
296
            accuracy = clf.score(x_test, y_test)*100
298         print("Depth: ", depth, " Accuracy: ", accuracy)

300     elif(val == 3):
            x_train, x_test, y_train, y_test = stratifiedOneHoldout
                                 (X_train, Y_train)
302
            if(depth == "None"):
304             clf = DecisionTreeClassifier(random_state = 0)
                clf.fit(X_train, Y_train)
306         else:
                clf = DecisionTreeClassifier(max_depth = depth,
                                 random_state = 0)
308             clf.fit(x_train, y_train)

310         accuracy = clf.score(x_test, y_test)*100
            print("Depth: ", depth, " Accuracy: ", accuracy)
312
        else:
314         print("Invalid validation tech.")

316 def kNN(k: int, metric: str, X_train, Y_train):

318     #Model
    if metric == "mahalanobis":
320       knn = KNeighborsClassifier(n_neighbors=k, weights='distance',
                                  metric=metric, algorithm="brute"
                                 , metric_params={'V': np.cov(
                                 X_train)})
    else:
322       knn = KNeighborsClassifier(n_neighbors=k, weights='distance',
                                  metric=metric)

324     #5-Fold
    cv_result_knn_5 = cross_val_score(knn, X_train, Y_train, cv=5,
                                 scoring='accuracy')
326
    #10-Fold
328 cv_result_knn_10 = cross_val_score(knn, X_train, Y_train, cv=10
                                 , scoring='accuracy')

330 #Random One Holdout
```

```
    x_train, x_test, y_train, y_test_random = randomOneHoldout(
                                    X_train, Y_train)
    knn.fit(x_train, y_train)

    y_pred_knn_random = knn.predict(x_test)

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

    print("5 Fold")
    print("KNN Accuracy: ", cv_result_knn_5.mean())

    print("10 Fold")
    print("KNN Accuracy: ", cv_result_knn_10.mean())

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

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

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)
```

```
368     y_pred_svm_random = svm.predict(x_test)

370     #Stratified One Holdout
        x_train, x_test, y_train, y_test_stratified =
                                    stratifiedOneHoldout(X_train,
                                    Y_train)
372     svm.fit(x_train, y_train)
        y_pred_svm_stratified = svm.predict(x_test)
374
        print("5 Fold")
376     print("SVM Accuracy: ", cv_result_svm_5.mean())

378     #print("10 Fold")
        #print("SVM Accuracy: ", cv_result_svm_10.mean())
380
        #print("Random One Hold Out")
382     #print("SVM Accuracy: ", 1 - metrics.mean_squared_error(
                                    y_test_random, y_pred_svm_random)
                                    )

384     #print("Stratified One Hold Out Fold")
        #print("SVM Accuracy: ", 1 - metrics.mean_squared_error(
                                    y_test_stratified,
                                    y_pred_svm_stratified))
386
    def zValues(df):
388
        cols = list(df.columns)
390     cols.remove('Index')

392     for col in cols:
            col_zscore = col + '_zscore'
394         df[col_zscore] = (df[col] - df[col].mean())/df[col].std(
                                    ddof=0)

396     return df

398 def outliers(df):
        return df.loc[(df._1_zscore > 2.5) | (df._2_zscore > 2.5) | (df
                                    ._2_zscore > 2.5) | (df._3_zscore
                                     > 2.5)]
400
    def subDatasets(df):
402     target0 = []
        target1 = []
404     target2 = []
        for row in df.itertuples():
406         if row.target == 0:
                target0.append(row)
408             target0_df = pd.DataFrame(target0)
```

```
        elif row.target == 1:
            target1.append(row)
            target1_df = pd.DataFrame(target1)
        else:
            target2.append(row)
            target2_df = pd.DataFrame(target2)

    dfs = [target0_df, target1_df, target2_df]

    for df in dfs:
        df.drop(columns=['target'])

    return target0_df, target1_df, target2_df

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=
                                    learning_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,
                                    random_state=0)
    clf.fit(X_train, Y_train)
    clf.predict(X_test)
    return clf.score(X_train, Y_train)

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,
                                    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")
```

```
446     print("KNN Accuracy: ", 1 - metrics.mean_squared_error(
                                    TrainDev_y, trainDev_pred))
        print( '\nClassification report\n' )
448     print(classification_report(y_true, trainDev_pred))

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

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

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

468 def displayAccuracy(X, Y):
        X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                    test_size=0.3, random_state=0)
470
        k = [3, 5, 7, 9, 11]
472     metric = ["euclidean", "manhattan", "chebyshev", "mahalanobis",
                                     "minkowski", "wminkowski", "
                                    seuclidean"]
        kernel = ["linear", "rbf"]
474     weight = [None, "balanced"]
        gamma = ["auto", "scale"]
476
        for i in k:
478       for j in metric:
              if j != "wminkowski" and j != "seuclidean":
480             print("K: {} - Metric: {}".format(i, j))
                kNN(i, j, X_train, Y_train)
482           print()

484     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()

if __name__ == '__main__':

  iris = datasets.load_iris()
  X = iris.data
  Y = iris.target

  #threeDVisualization(iris.data[:, :], Y)

  # Z-Score
  scaler = StandardScaler()
  scaler.fit(X)
  z_score = scaler.transform(X)

  #Displaying result according to each type of methods and
                                regression model
  print("\nRaw: ")
  displayAccuracy(X,Y)
  #print("\nZ-Score: ")
  #displayAccuracy(z_score,Y)

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

    X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                test_size=0.2, random_state=0)

    #svm.fit(X_train, Y_train)
    #y_true, y_pred = Y_test, svm.predict(X_test)
    #print("SVM Accuracy: ", 1 - metrics.mean_squared_error(Y_test,
                                 y_pred))

    fourError(X, Y, svm)

    #AdaBoost(svm, 100, 1, X_train, Y_train, X_test, Y_test)
    #GradientBoost(100, 1, X_train, Y_train, X_test, Y_test)

```

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

    knn = KNeighborsClassifier(n_neighbors=7, weights='uniform',
                                   metric="chebyshev")
    #knn.fit(X_train, Y_train)

    #y_true, y_pred = Y_test, knn.predict(X_test)
    fourError(X, Y, knn)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size
                                   =0.2, random_state=9)
#Worst model
complement = ComplementNB()
learning_rate = [0.0001, 0.001, 0.01, 0.1, 1 ,2 ,3 ,4, 5]

result_ada_1 = []
result_gradient_1 = []
for i in learning_rate: #i -> Learning Rate
    x = []
    y = []
    for j in range(50, 150): #j -> N estimators
        x.append(AdaBoost(complement, j, i, X_train, Y_train,
                                   X_test, Y_test))
        y.append(GradientBoost(j, i, X_train, Y_train, X_test,
                                   Y_test))
    result_ada_1.append(x)
    result_gradient_1.append(y)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size
                                   =0.2, random_state=0)
#best model
clf = DecisionTreeClassifier(max_depth = 3)
learning_rate = [0.0001, 0.001, 0.01, 0.1, 1 ,2 ,3 ,4, 5]

result_ada_2 = []
result_gradient_2 = []
for i in learning_rate: #i -> Learning Rate
    x = []
    y = []
    for j in range(50, 150): #j -> N estimators
        x.append(AdaBoost(clf, j, i, X_train, Y_train, X_test,
                                   Y_test))
        y.append(GradientBoost(j, i, X_train, Y_train, X_test,
                                   Y_test))
    result_ada_2.append(x)
    result_gradient_2.append(y)

from pandas.tools.plotting import parallel_coordinates
# Perform parallel coordinate plot
```

```
parallel_coordinates(result_ada_1, 'Class')
plt.show()

parallel_coordinates(result_ada_2, 'Class')
plt.show()

parallel_coordinates(result_gradient_1, 'Class')
plt.show()

parallel_coordinates(result_gradient_2, 'Class')
plt.show()
```