PROJECT REPORT

Canberk Arıcı – 171044062

***** Dear madam, summary of the article is at the end of the report. ******

Project Name: Parkinson's Disease Classification
Demo Link: https://youtu.be/y1mPEjVTtuE
(Demo restriction is 4 minutes but I couldn't fit all informations I need to mention into 4 minutes therefore it's more than 4 minutes, I'm sorry for that.)
Dataset Link: **https://archive.ics.uci.edu/ml/datasets/Parkinson%27s+Disease+Classification**
*** Dataset includes 754 columns and each column has 756 instances.

Data Cleaning(Cleaning Useless Data):

```
del df['id']
```

I drop "id" feature since every value of it is unique and it does not contribute to classification.

Feature Extraction

There are totally 754 features in the dataset and it's too much therefore I find highly correlated features and drop them in order to decrease number of dimensions. I firstly find matrix of absolute value of correlation between features then I get its upper triangle since lower and upper triangles are the same and diagonal is intersection of the same features. Finally I select features which are having correlation greater than 0.95 and drop them.

Implementation:

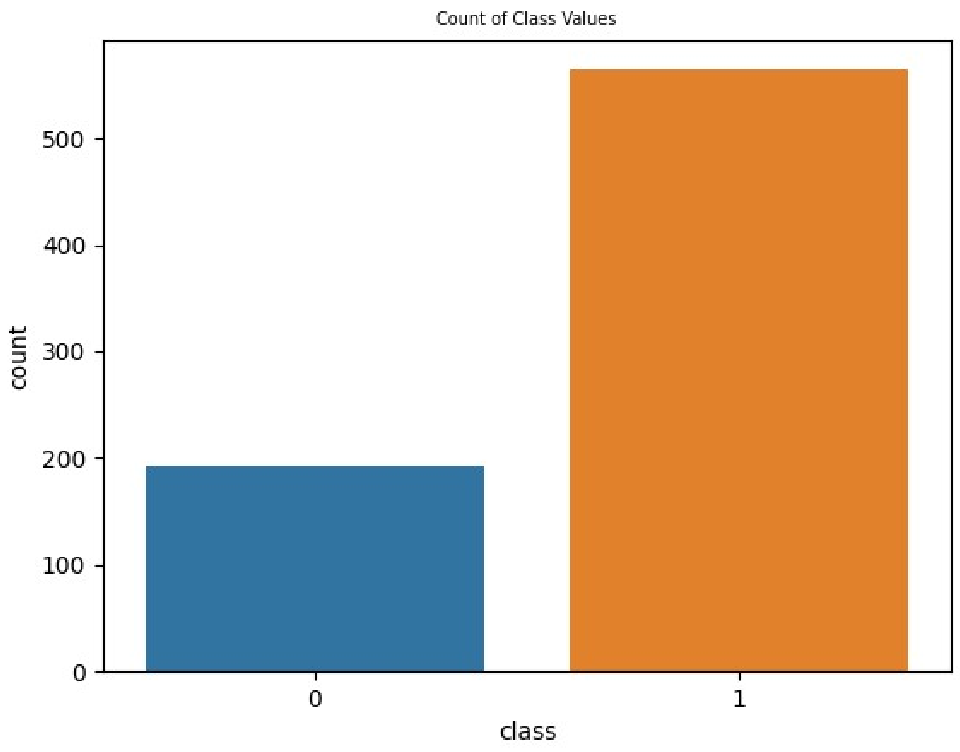
```
def drop_highly_correlated(df):
    # I will have the elements as the absolute value of correlation between the features
    cor_matrix = df.corr().abs()
    # I take upper triangle because upper and lower triangle's are the same
    upper_tri = cor_matrix.where(np.triu(np.ones(cor_matrix.shape),k=1).astype(np.bool))
    # I am selecting the columns which are having absolute correlation greater than 0.95
    # and creating a list of those columns
    to_drop = [column for column in upper_tri.columns if any(upper_tri[column] > 0.95)]
    print()
    print("Number of features in df BEFORE drop: {}".format(len(df.columns)))
    print()
    print("Number of features that will be dropped: {}".format(len(to_drop)))
    print()
    df = df.drop(to_drop, axis=1)
    print("Number of features in df AFTER drop: {}".format(len(df.columns)))
    print()
    print(df.head())
    return df

df = drop_highly_correlated(df)
```

```
Number of features in df BEFORE drop: 754

Number of features that will be dropped: 241

Number of features in df AFTER drop: 513
```

Since number of features is not appropriate for visualization, I couldn't visualize dataset as much as I want.

Balance of Target Feature

If it's balanced then we can get good results for classification.



There is difference but we can still get good results.

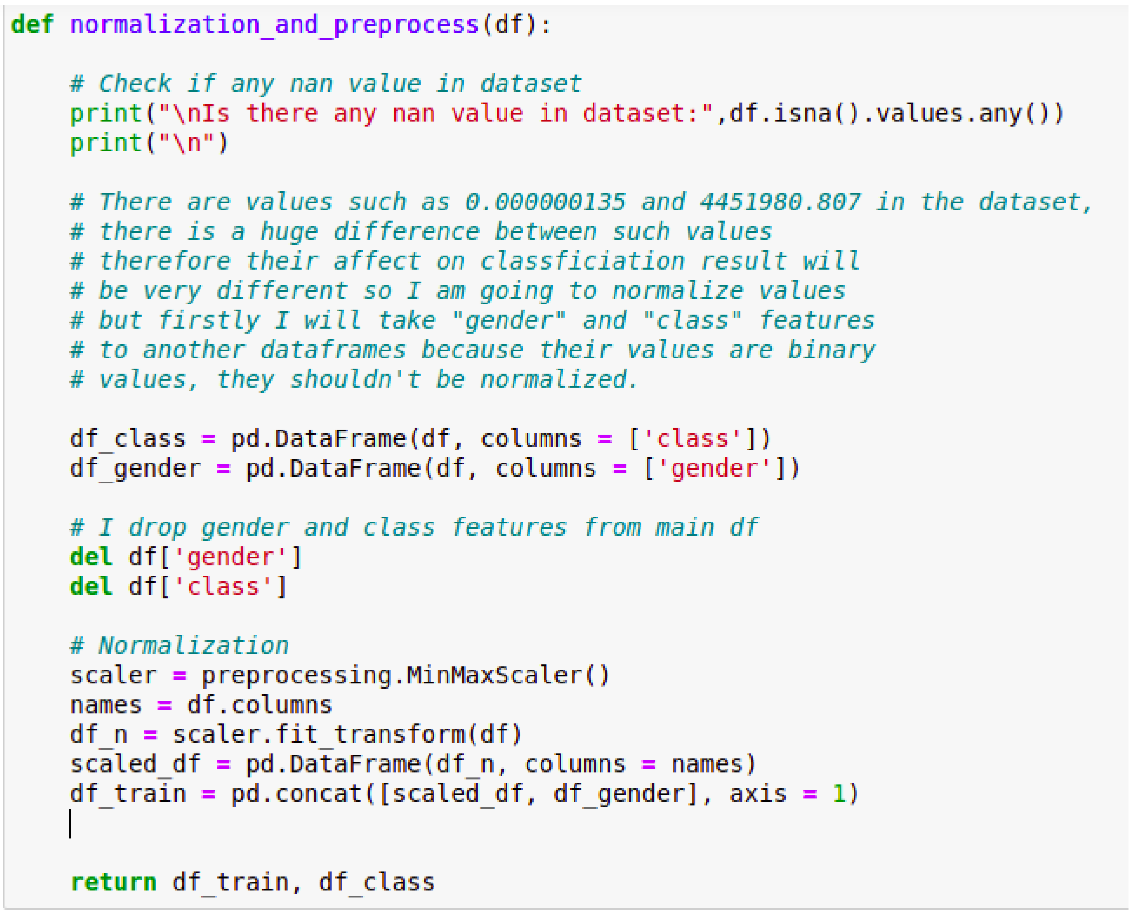
***** There is no categorical value in dataset therefore no need to turn them into numerical values.**

NORMALIZATION

There are values such as 0.000000135 and 4451980.807 in the dataset, there is a huge difference between such values therefore their affect on classficiation result will be very different so I am going to normalize values but firstly I will take "gender" and "class" features to another dataframes because their values are binary values, they shouldn't be normalized.

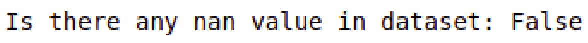
I used MinMaxScaler() because it fits data values between 0 and 1.

Normalization Implementation:



```python
def normalization_and_preprocess(df):

    # Check if any nan value in dataset
    print("\nIs there any nan value in dataset:",df.isna().values.any())
    print("\n")

    # There are values such as 0.000000135 and 4451980.807 in the dataset,
    # there is a huge difference between such values
    # therefore their affect on classficiation result will
    # be very different so I am going to normalize values
    # but firstly I will take "gender" and "class" features
    # to another dataframes because their values are binary
    # values, they shouldn't be normalized.

    df_class = pd.DataFrame(df, columns = ['class'])
    df_gender = pd.DataFrame(df, columns = ['gender'])

    # I drop gender and class features from main df
    del df['gender']
    del df['class']

    # Normalization
    scaler = preprocessing.MinMaxScaler()
    names = df.columns
    df_n = scaler.fit_transform(df)
    scaled_df = pd.DataFrame(df_n, columns = names)
    df_train = pd.concat([scaled_df, df_gender], axis = 1)


    return df_train, df_class
```

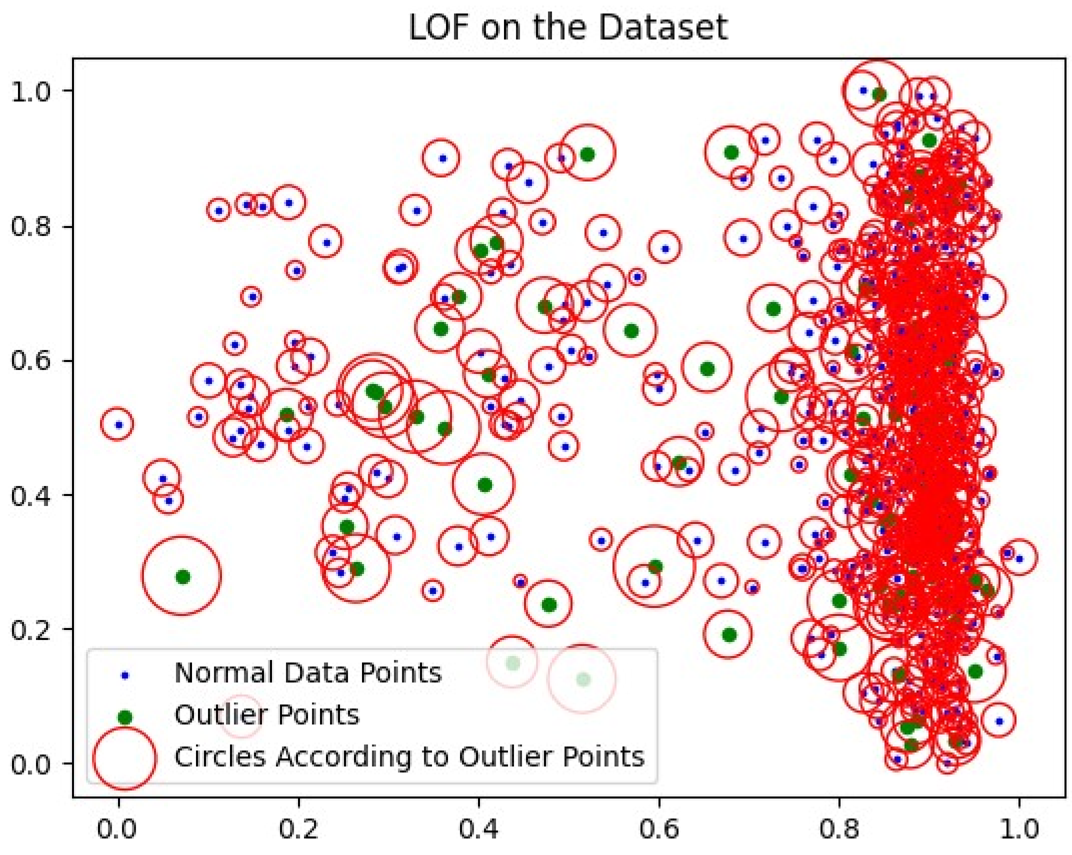
Also I check for whether there is any nan value in the dataset.



```
Is there any nan value in dataset: False
```

Outlier Detection with Local Outlier Factor (LOF)

I detected and dropped outliers by using LOF. I applied data by using fit_predict to LOF and then I kept outlier scores into a new dataframe. I selected a threshold value which very discrete from other values in the dataset and it is -1.25, by selecting threshold as -1.25, I get indexes of outlier values which are having score greater than 1.25. Here is the graphic of outliers in dataset:



Implementation:

```
columns = X.columns.tolist()

# 'auto' will attempt to decide the most appropriate algorithm
# to compute the nearest neighbors based on the values passed to fit method.
lof = LocalOutlierFactor(algorithm = 'auto')
pred_values = lof.fit_predict(X)
scores_of_X = lof.negative_outlier_factor_

# I create dataframe to keep outlier scores and then to get indexes of them
df_outlier = pd.DataFrame()
df_outlier['scores'] = scores_of_X
```

```
# I selected threshold as -1.25 and I get indexes of outlier
# values which are having score greater than 1.25.
index_of_outliers = df_outlier[df_outlier['scores'] < -1.25].index.tolist()

plt.figure()
plt.title("LOF on the Dataset")
plt.scatter(X.iloc[:, 0], X.iloc[:, 1], color="b", s=3.0, label="Normal Data Points")
plt.scatter(X.iloc[index_of_outliers, 0], X.iloc[index_of_outliers, 1], color="g", s=20.0, label="Outlier Points")

# Normalizing in order to see plot of outliers
radius_of_circles = (scores_of_X.max() - scores_of_X) / (scores_of_X.max() - scores_of_X.min())
plt.scatter(X.iloc[:, 0], X.iloc[:, 1], s=1000 * radius_of_circles, edgecolors="r", facecolors="none",
    label="Circles According to Outlier Points",
)
plt.legend()
plt.show()
```

Now I drop the outliers from the dataset:

```
X = X.drop(index_of_outliers)
Y = Y.drop(index_of_outliers).values
```

PCA

I use PCA to reduce the dimensionality of my dataset, I want to reduce dimensionality by taking only the dimensions with the highest importance and those newly fine dimensions will minimize the projection error and the projected points will have a maximum spread or which means the maximum variance. Via the maximum spread, projection error will be minimal and I can contain most of the information about the data.

PCA steps I have applied are given below:
- Subtracting the mean from X
- Calculating Cov(X,X)

Here is Mean and Covariance Matrix Formulas:

$$\hat{\boldsymbol{\mu}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i$$

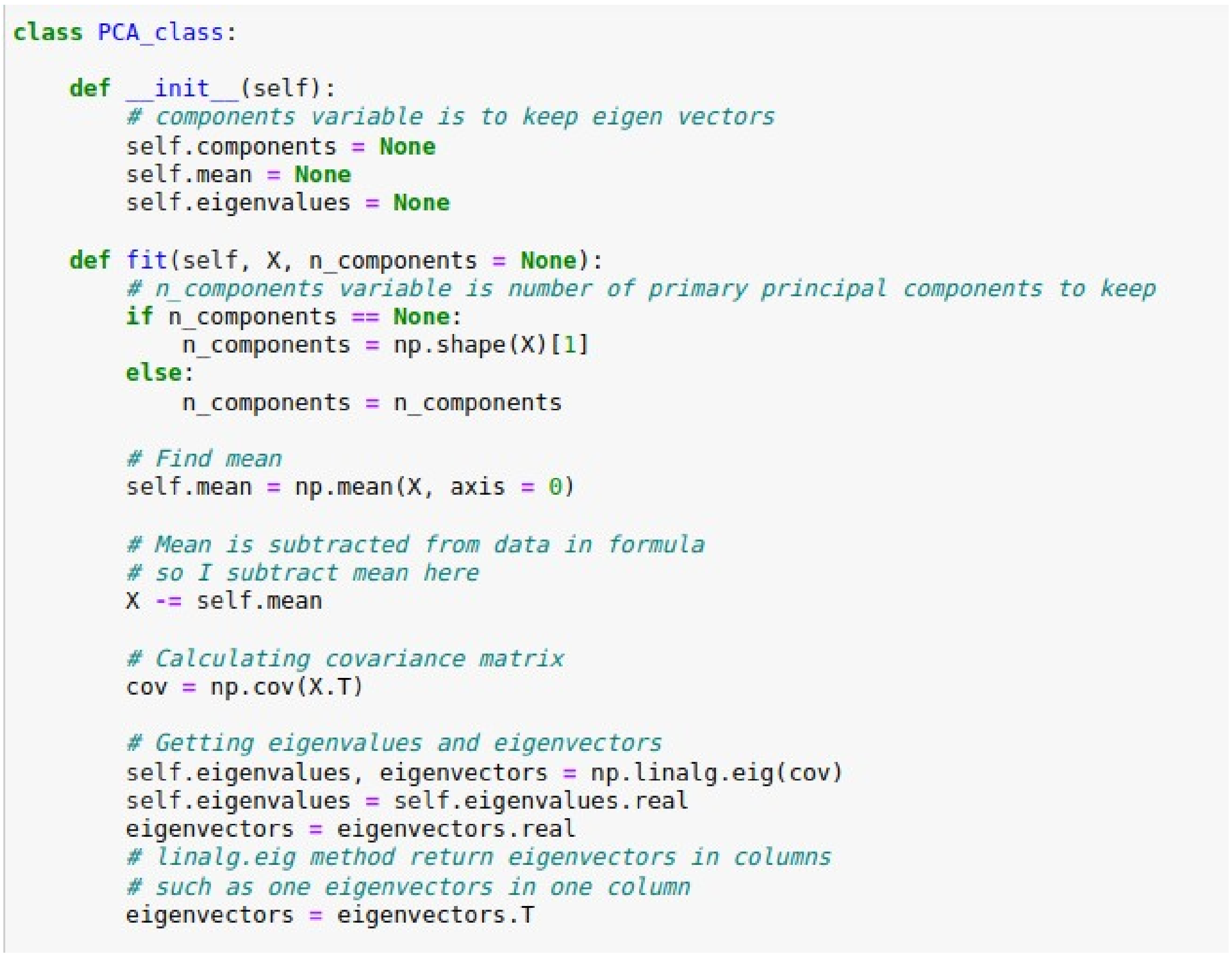
$$\hat{\Sigma} = \frac{1}{N}\sum_{i=1}^{N} (\mathbf{x}_i - \hat{\boldsymbol{\mu}})(\mathbf{x}_i - \hat{\boldsymbol{\mu}})^\top$$

(source of image: https://stackoverflow.com/questions/38571944/calculate-covariance-matrix-formula)

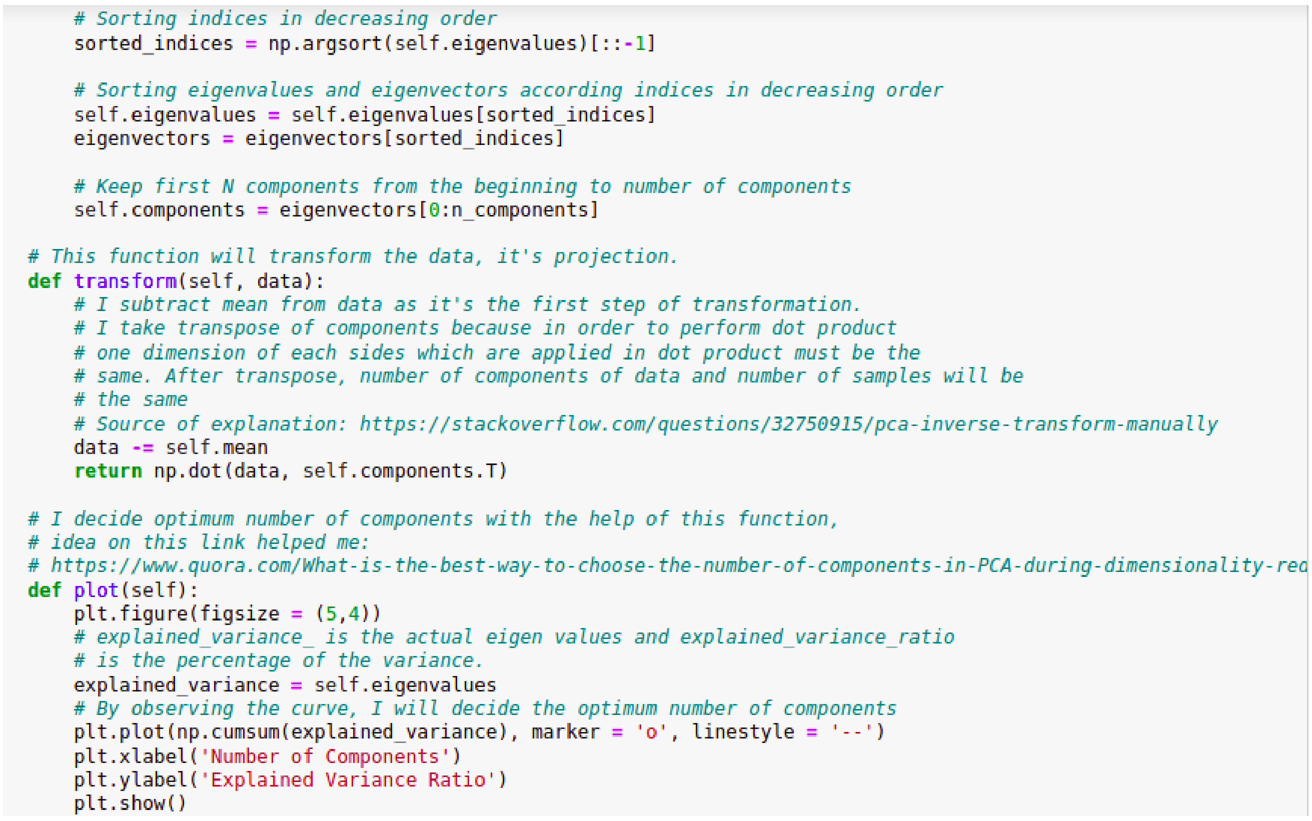
- Calculating eigenvectors and eigenvalues of covariance matrix
- Sorting eigenvectors according to their eigenvalues in decreasing order
- Choosing first k eigenvectors which will be the new k dimensions
- Transform the original n dimensional data into k dimensional data by projecting them which is the dot product

I have written PCA from scratch.

PCA Implementation:



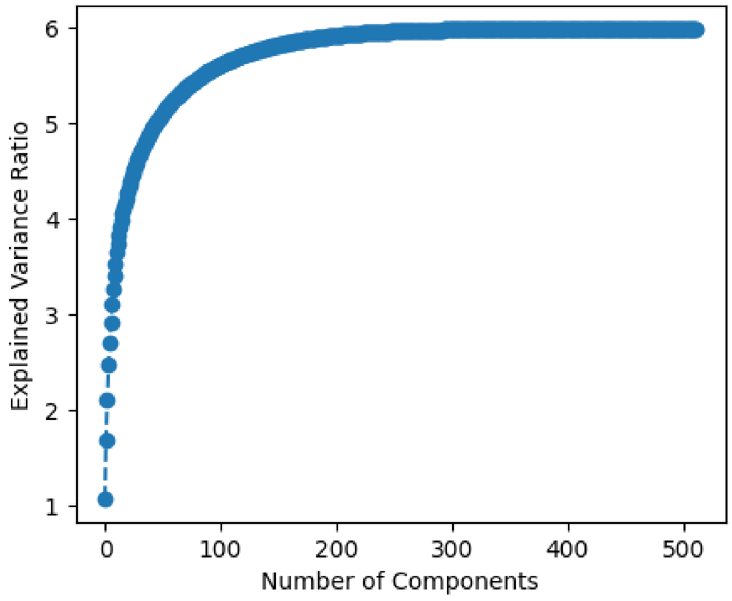
```python
class PCA_class:

    def __init__(self):
        # components variable is to keep eigen vectors
        self.components = None
        self.mean = None
        self.eigenvalues = None

    def fit(self, X, n_components = None):
        # n_components variable is number of primary principal components to keep
        if n_components == None:
            n_components = np.shape(X)[1]
        else:
            n_components = n_components

        # Find mean
        self.mean = np.mean(X, axis = 0)

        # Mean is subtracted from data in formula
        # so I subtract mean here
        X -= self.mean

        # Calculating covariance matrix
        cov = np.cov(X.T)

        # Getting eigenvalues and eigenvectors
        self.eigenvalues, eigenvectors = np.linalg.eig(cov)
        self.eigenvalues = self.eigenvalues.real
        eigenvectors = eigenvectors.real
        # linalg.eig method return eigenvectors in columns
        # such as one eigenvectors in one column
        eigenvectors = eigenvectors.T
```



```python
        # Sorting indices in decreasing order
        sorted_indices = np.argsort(self.eigenvalues)[::-1]

        # Sorting eigenvalues and eigenvectors according indices in decreasing order
        self.eigenvalues = self.eigenvalues[sorted_indices]
        eigenvectors = eigenvectors[sorted_indices]

        # Keep first N components from the beginning to number of components
        self.components = eigenvectors[0:n_components]

    # This function will transform the data, it's projection.
    def transform(self, data):
        # I subtract mean from data as it's the first step of transformation.
        # I take transpose of components because in order to perform dot product
        # one dimension of each sides which are applied in dot product must be the
        # same. After transpose, number of components of data and number of samples will be
        # the same
        # Source of explanation: https://stackoverflow.com/questions/32750915/pca-inverse-transform-manually
        data -= self.mean
        return np.dot(data, self.components.T)

    # I decide optimum number of components with the help of this function,
    # idea on this link helped me:
    # https://www.quora.com/What-is-the-best-way-to-choose-the-number-of-components-in-PCA-during-dimensionality-red
    def plot(self):
        plt.figure(figsize = (5,4))
        # explained_variance_ is the actual eigen values and explained_variance_ratio
        # is the percentage of the variance.
        explained_variance = self.eigenvalues
        # By observing the curve, I will decide the optimum number of components
        plt.plot(np.cumsum(explained_variance), marker = 'o', linestyle = '--')
        plt.xlabel('Number of Components')
        plt.ylabel('Explained Variance Ratio')
        plt.show()
```

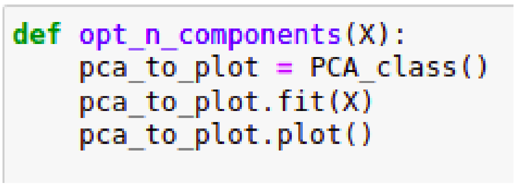
Determining Optimum Number of Components for PCA

I plot the curve according to Cumulative Sum of Explained Variance(Eigenvalues) to determine optimum number of components in order to keep %90 of variation. The graphic in jupyter-notebook is interactive so we can see what percentage of data is kept at which number of components.



As shown in the interactive curve above, the first 42 components describe about 80% of the variance, whereas we'll need roughly 121 components to describe close to 100% of the variance. Since the 2D projection loses a lot of information, we'll need roughly 60 components to keep 90% of the variation.

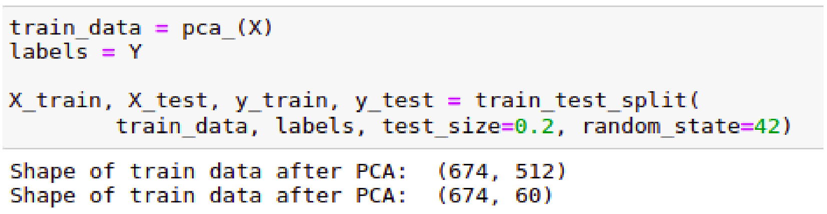
Implementation:



```
def opt_n_components(X):
    pca_to_plot = PCA_class()
    pca_to_plot.fit(X)
    pca_to_plot.plot()
```

Train Test Split

I split my dataset into 4 parts: x_train, x_test, y_train, y_test.



```
train_data = pca_(X)
labels = Y

X_train, X_test, y_train, y_test = train_test_split(
        train_data, labels, test_size=0.2, random_state=42)
```

```
Shape of train data after PCA:  (674, 512)
Shape of train data after PCA:  (674, 60)
```

***** We can see the result of PCA here.**

CLASSIFIERS AND RESULTS

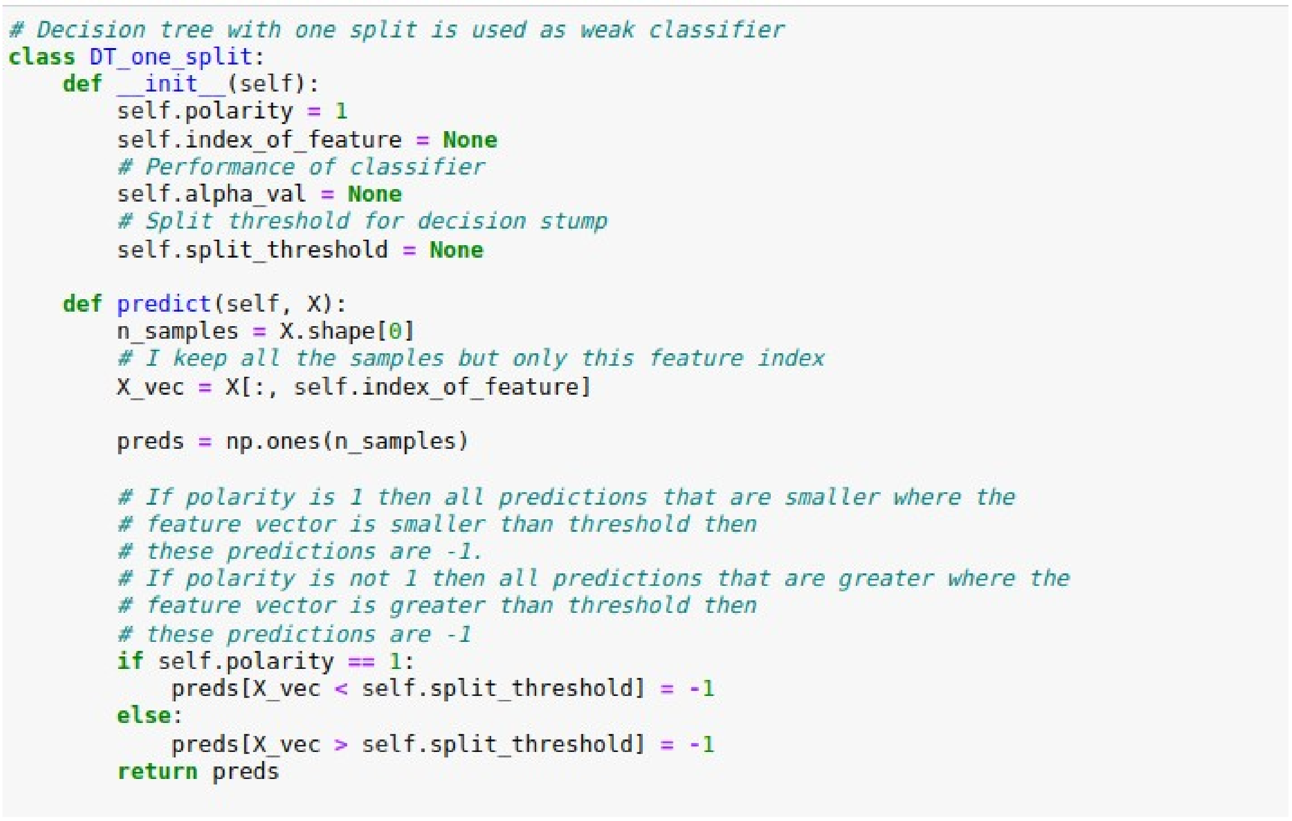
I used SVM, RandomForest and MLPClassifier methods as built-in and I implemented AdaBoost from scratch then use them for classification. We didn't mention about AdaBoost that much in the lesson and I was impressed when I knew details of AdaBoost therefore I have chosen to implement AdaBoost. I researched it in detail and learned a lot about it's background then implemented it from scratch. I compared accuracy of my implementation and built-in AdaBoost.

SVM: Each data item is plotted as a point in n-dimensional space (where n is the number of features you have), with the value of each feature being the value of a certain coordinate in the SVM algorithm. Then we accomplish classification by locating the hyper-plane that clearly distinguishes the two classes.

RandomForest: The random forest is a decision tree-based classification technique. When constructing each individual tree, it employs bagging and feature randomization in order to generate an uncorrelated forest of trees whose committee forecast is more accurate than any single tree's.

MLPClassifier: MLPClassifier is an acronym for Multi-layer Perceptron Classifier, which is linked to a Neural Network by its name. Unlike other classification algorithms like Support Vectors or Naive Bayes Classifier, MLPClassifier does classification using an underlying Neural Network.

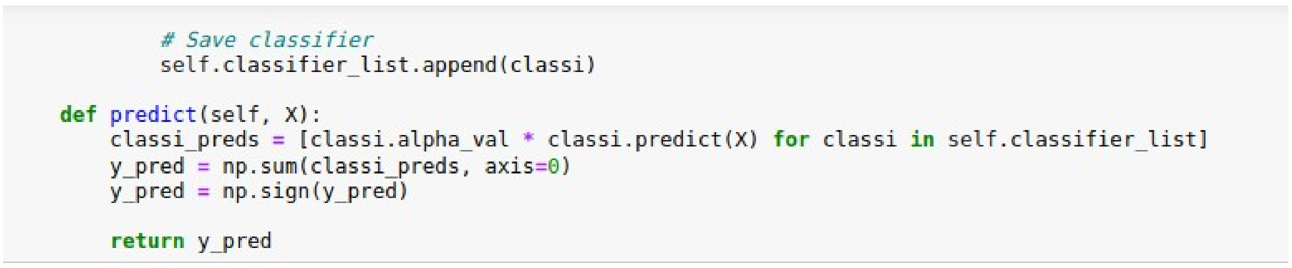
AdaBoost Implementation:



```
# Decision tree with one split is used as weak classifier
class DT_one_split:
    def __init__(self):
        self.polarity = 1
        self.index_of_feature = None
        # Performance of classifier
        self.alpha_val = None
        # Split threshold for decision stump
        self.split_threshold = None

    def predict(self, X):
        n_samples = X.shape[0]
        # I keep all the samples but only this feature index
        X_vec = X[:, self.index_of_feature]

        preds = np.ones(n_samples)

        # If polarity is 1 then all predictions that are smaller where the
        # feature vector is smaller than threshold then
        # these predictions are -1.
        # If polarity is not 1 then all predictions that are greater where the
        # feature vector is greater than threshold then
        # these predictions are -1
        if self.polarity == 1:
            preds[X_vec < self.split_threshold] = -1
        else:
            preds[X_vec > self.split_threshold] = -1
        return preds
```

```python
class AdaBoost_m:
    def __init__(self, num_of_classifiers=5):
        # number of classifiers
        self.num_of_classifiers = num_of_classifiers
        self.classifier_list = []

    def fit(self, X, y):
        num_samples, num_features = X.shape

        # init weights to 1/N
        w = np.full(num_samples, (1 / num_samples))

        # List to store all classifiers
        self.classifier_list = []

        # Iterate through classifiers
        for _ in range(self.num_of_classifiers):
            classi = DT_one_split()
            # I want to find the best split feature value and best split threshold
            # where this error is minimum
            error_min = float("inf")

            # Iterate over features and thresholds
            # in order to find best threshold and feature
            for ind_feature in range(num_features):
                X_vec = X[:, ind_feature]
                # I get unique values and they are thresholds
                thresholds = np.unique(X_vec)

                for threshold in thresholds:
                    p = 1
                    preds = np.ones(num_samples)
                    preds[X_vec < threshold] = -1
```

```python
                    # Error = sum of weights over misclassified samples
                    preds = preds.reshape(len(preds),1)
                    w = w.reshape(len(w),1)
                    misclassified = w[y != preds]
                    error = sum(misclassified)

                    if error > 0.5:
                        error = 1 - error
                        p = -1

                    # Keeping the best configuration
                    if error < error_min:
                        classi.split_threshold = threshold
                        classi.polarity = p
                        classi.index_of_feature = ind_feature
                        error_min = error

            # calculate alpha
            # EPS is used in order to prevent division by zero, EPS is a very small value.
            EPS = 1e-10
            # Formula to find alpha is applied below.
            classi.alpha_val = 0.5 * np.log((1.0 - error_min + EPS) / (error_min + EPS))

            # Calculate predictions and update weights
            preds = classi.predict(X)
            preds = preds.reshape(len(preds),1)

            # Here I update weights
            w *= np.exp(-classi.alpha_val * y * preds)

            # Normalize to one
            w /= np.sum(w)
```



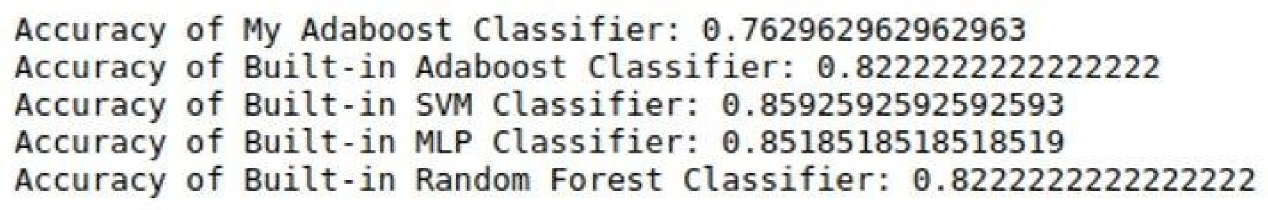
```
            # Save classifier
            self.classifier_list.append(classi)

    def predict(self, X):
        classi_preds = [classi.alpha_val * classi.predict(X) for classi in self.classifier_list]
        y_pred = np.sum(classi_preds, axis=0)
        y_pred = np.sign(y_pred)

        return y_pred
```

Algorithm of AdaBoost:

```
- Initialize weights to 1/N for each sample
for t in T: (where T is number of weak learner)
    - Train weak classifier in order to find the best split feature and the best split threshold
    - Calculate error for decision tree with one split
        - If error sum is greater than 0.5 then flip error (1-error) and decision
    - Calculate performance of classifier (alpha_val)
    - Update weights
```

Mathematical Equations of Adaboost:
https://towardsdatascience.com/adaboost-for-dummies-breaking-down-the-math-and-its-equations-into-simple-terms-87f439757dcf

RESULTS



```
Accuracy of My Adaboost Classifier: 0.762962962962963
Accuracy of Built-in Adaboost Classifier: 0.8222222222222222
Accuracy of Built-in SVM Classifier: 0.8592592592592593
Accuracy of Built-in MLP Classifier: 0.8518518518518519
Accuracy of Built-in Random Forest Classifier: 0.8222222222222222
```

SUMMARY OF ARTICLE

Computer-aided identification of degenerative neuromuscular diseases based on gait dynamics and ensemble decision tree classifiers

Luay Fraiwan, Omnia Hassanin

Published: June 4, 2021 • https://doi.org/10.1371/journal.pone.0252380

All muscles, sensory neurons, and motor neurons are part of the neuromuscular system, which controls human mobility. The deterioration or gradual loss of function in efferent or afferent neurons causes degenerative neuromuscular disorders (DNDs). Afferent nerves convey sensory information back to the brain and central nervous system, whereas efferent nerves regulate voluntary muscles. Amyotrophic lateral sclerosis (ALS), Parkinson's disease (PD), and Huntington's disease are just a few examples of frequent DNDs (HD).
The use of computer-assisted human locomotion analysis to identify DNDs has lately garnered substantial scientific interest, driven by the demand for cost-effective, non-invasive clinical treatments. Artificial intelligence algorithms are commonly used in computer-aided diagnosis systems. The aspects of human knowledge and mistake become less destructive to the diagnosis process if the data is collected and processed correctly, and the detection algorithm is well-chosen and tuned.

Bagging, AdaBoost, RUSBoost, and random subspace were among the DT ensemble versions that were used for classification. The multi-class classification problem was solved by assembling a one-versus-all error-correcting output code. The models are trained and evaluated using 10-fold cross-validation to provide a reliable estimate of overall classification performance. The folds are separated using an equi-stratified method to adjust for data imbalance. A consistent sample size was maintained between illness levels for each feature class. This method combines the predictions of numerous base classifiers to make a multi-class classification decision. Each base classifier carries out a single binary classification job aimed at distinguishing one class from the others.

Ensemble classifier systems outperform their constituent base models, according to the classification results. The VGRF-based features set had the greatest classification performance when used with the base decision tree model, while other ensemble strategies improved classification results to variable degrees. All metrics improved significantly with the AdaBoost.

Bagging and RUSBoost models were shown to provide slightly lesser performance increases. Assisting in smart long-term monitoring is one of the potential benefits of such an accurate diagnostic system. This also provides physicians and health-care professionals with noninvasive and low-cost diagnostic tools. The small-length raw VGRF signals utilized to construct the stride, stance, and swing parameters signals might explain the reduced accuracies. However, because the provided dataset was insufficient for multiple fold validation, this method was used.

To find the best appropriate parameter for all classification algorithms, a data-driven hyperparameter tuning strategy using Bayesian optimization was used. As a consequence of the promising findings in recognizing frequent DNDs, the suggested framework may be used to assist in diagnostic choices in computing hardware resource-constrained contexts.