Senih Yılmaz & Mert Umurhan
21802154 & 21802142
09/04/2022

EEE 485-1 FINAL REPORT: HEART DISEASE PREDICTION

Project Description

The World Health Organization states that "Cardiovascular diseases (CVDs) are the leading cause of death globally" [1]. Computer-aided detection of such diseases could make the detection faster and more accurate. In this paper, we have written a machine learning program to detect Cardiovascular diseases from given features. For this purpose, we used a cardiovascular disease dataset that is taken from Kaggle. The machine learning models that we mentioned in the proposal are trained with this dataset to detect whether a patient will have cardiovascular disease by stating its probability. We have written three supervised learning models: logistic regression, support vector machine, and decision tree from scratch. Logistic regression is used to model the probability of the given parameters to the outcome. Support vector machine is a learning model that uses its instances as vectors and classifies them using hyperplanes. Decision tree is a learning model that depends on an impurity metric that creates a structure where each parameter split will create branches of the tree and the dataset will shape the branches and probabilities of the tree [2]. We used Python as our programming language since it is recommended and well-suited for machine learning projects.

Dataset

The dataset cardio is a CSV file that contains 12 features and 70000 data instances about medical records that are useful for cardiovascular disease detection. Having 70000 data instances should be beneficial for training an accurate model [3]. The features include:

1. Age | Objective Feature | age | int (days) > converted into (years)
2. Height | Objective Feature | height | int (cm) |
3. Weight | Objective Feature | weight | float (kg) |
4. Gender | Objective Feature | gender | categorical code |
5. Systolic blood pressure | Examination Feature | ap_hi | int |
6. Diastolic blood pressure | Examination Feature | ap_lo | int |
7. Cholesterol | Examination Feature | cholesterol | 1: normal, 2: above normal, 3: well above normal |
8. Glucose | Examination Feature | gluc | 1: normal, 2: above normal, 3: well above normal |
9. Smoking | Subjective Feature | smoke | binary |
10. Alcohol intake | Subjective Feature | alco | binary |
11. Physical activity | Subjective Feature | active | binary |
12. Presence or absence of cardiovascular disease | Target Variable | cardio | binary |

records of the patients. Moreover, there is a column named “id” in the dataset which will not give any explanatory information about the cardiovascular disease probability of the patient, therefore, we have excluded that column (intentionally not referring to it as “feature”).

TABLE I: SCREENSHOT FROM THE DATASET

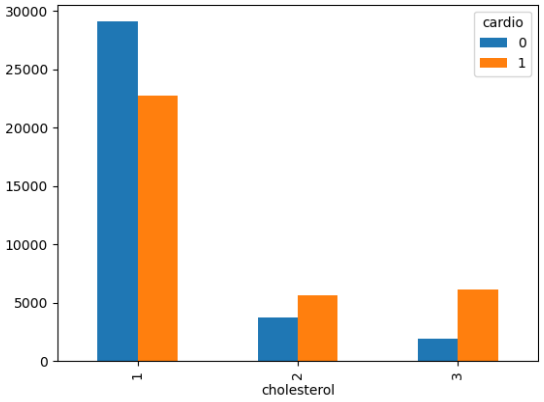
	A	B	C	D	E	F	G	H	I	J	K	L	M
1	id	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio
2	0	18393	2	168	62	110	80	1	1	0	0	1	0
3	1	20228	1	156	85	140	90	3	1	0	0	1	1
4	2	18857	1	165	64	130	70	3	1	0	0	0	1
5	3	17623	2	169	82	150	100	1	1	0	0	1	1
6	4	17474	1	156	56	100	60	1	1	0	0	0	0
7	8	21914	1	151	67	120	80	2	2	0	0	0	0
8	9	22113	1	157	93	130	80	3	1	0	0	1	0
9	12	22584	2	178	95	130	90	3	3	0	0	1	1
10	13	17668	1	158	71	110	70	1	1	0	0	1	0
11	14	19834	1	164	68	110	60	1	1	0	0	0	0
12	15	22530	1	169	80	120	80	1	1	0	0	1	0
13	16	18815	2	173	60	120	80	1	1	0	0	1	0
14	18	14791	2	165	60	120	80	1	1	0	0	0	0
15	21	19809	1	158	78	110	70	1	1	0	0	1	0
16	23	14532	2	181	95	130	90	1	1	1	1	1	0
17	24	16782	2	172	112	120	80	1	1	0	0	0	1
18	25	21296	1	170	75	130	70	1	1	0	0	0	0
19	27	16747	1	158	52	110	70	1	3	0	0	1	0
20	28	17482	1	154	68	100	70	1	1	0	0	0	0
21	29	21755	2	162	56	120	70	1	1	1	0	1	0
22	30	19778	2	163	83	120	80	1	1	0	0	1	0
23	31	21413	1	157	69	130	80	1	1	0	0	1	0

cardio_train

Above there is a screenshot of our data to give a quick look at what our data looks like.

Preprocessing

Initially, we converted the age feature from days to years to get a better intuition. Secondly, 5 number summary (min 25% 50% 75% max), mean and standard deviation values were inspected as the summary of the features. Before conducting any filtration, our dataset had 70000 instances. Having researched about the topic, we concluded that some filtrations on the data should be made. For instance, having a systolic blood pressure lower than 50 or higher than 1000 is extremely rare, even it could be fatal [4]. Similarly, diastolic blood pressure has a tolerance range that we have researched about [5]. Moreover, there were some unlikely height and weight values in the dataset. Thus, removing them in the preprocess stage enabled us to clean our data. Thirdly, we plotted some graphs to see if there were any correlation between response (cardio) and features. Summarizing the data again, we realized that we eliminated around 700 instances, which is only 1% of the initial data. As we expected, there were some lessons to be learned from the features of the cleaned dataset even by looking at their simple visualizations (bar charts). Finally, we moved on to the implementation of our proposed models to predict the response using the features.



For some methods we also had to normalize the data. To normalize the data:

$$data_i = \frac{data_i - mean(data_i)}{standard\,deviation(data_i)}$$

Where each i is a different parameter.

Description and Implementation of Methods

1. **Logistic Regression**

This method uses a logistic function to map a binary dependent probability measure. Since this project's goal is to do a binary classification, logistic regression is a preferable method. Using the probability measure, binary classification can be done via a chosen threshold.[6]

Logistic function is:

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + \sum_i \beta_i x_i)}}$$

Where:

$$\beta_0 + \sum_i \beta_i x_i \mapsto \infty$$

then p(x) = 1

$$\beta_0 + \sum_i \beta_i x_i \mapsto -\infty$$

then p(x) = 0

Loss function for this regression is:

$$L = \prod_{k:y_k=1} p_k \prod_{k:y_k=0} (1 - p_k)$$

Where k is the one instance. Log-loss function can be found as:

$$\ell = \sum_{k:y_k=1} \ln(p_k) + \sum_{k:y_k=0} \ln(1-p_k) = \sum_{k=1}^{K} \left(y_k \ln(p_k) + (1-y_k) \ln(1-p_k) \right)$$

To minimize the log-loss function, We've used gradient descent:

$$\beta_{new} = \beta_{old} - \gamma \nabla L$$

Where γ is the learning rate. Gradient of L (∇L) is:

$$\nabla L = \frac{1}{N} \sum_{k} (p_k - y_k) x_k$$

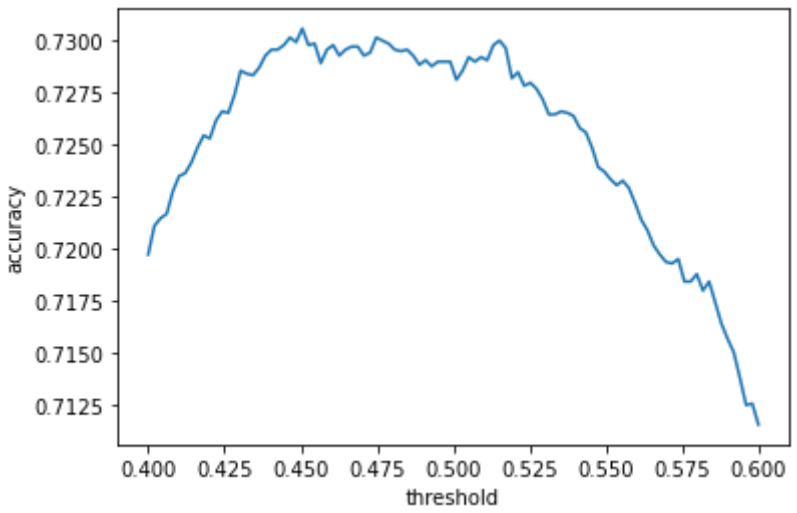
We've iterated the gradient descent until loss change is lower than a selected threshold. This part was advised in the project demo since we were iterating the gradient descent a determined amount of time. Now it iterates until the loss converges.

To apply the trained β 's on a test set we use the p(x) function:

$$y_{guess,k} = 1 \text{ if } p(x_{test,k}) > threshold$$

$$y_{guess,k} = 0 \text{ if } p(x_{test,k}) \leq threshold$$

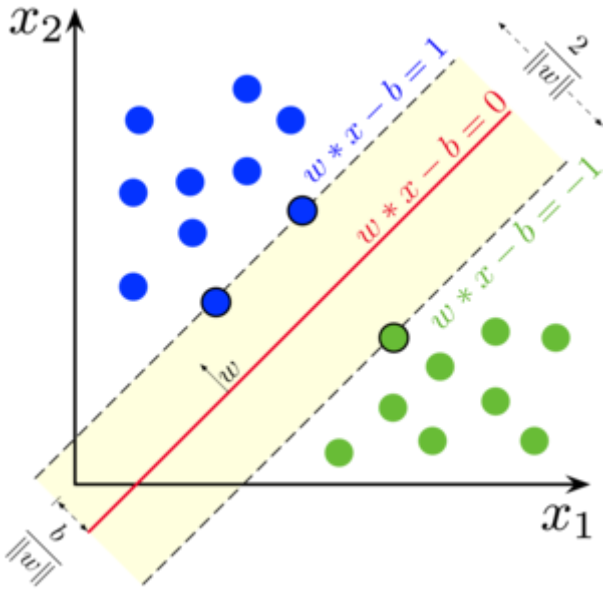
The model is trained using 60% of the data set. Next we've used the validation set (%20) to determine the best threshold.



Where maximum accuracy is ~%73 and the threshold is 0.45. Then we've used the test set (%20) to find its accuracy using this threshold, which is ~%72.45.

2. Support Vector Machine

Support Vector Machine (SVM) method takes each instance as a vector and fits a hyperplane according to the training set. It fits the hyperplane so that the dot product of an instance and normal vector of the hyperplane can be used for classification. SVM's can be linear or non-linear by using the Kernel trick. Below is a SVM example with hyperplane (red) and support vectors.[7]



Non-Linear SVM Trials:

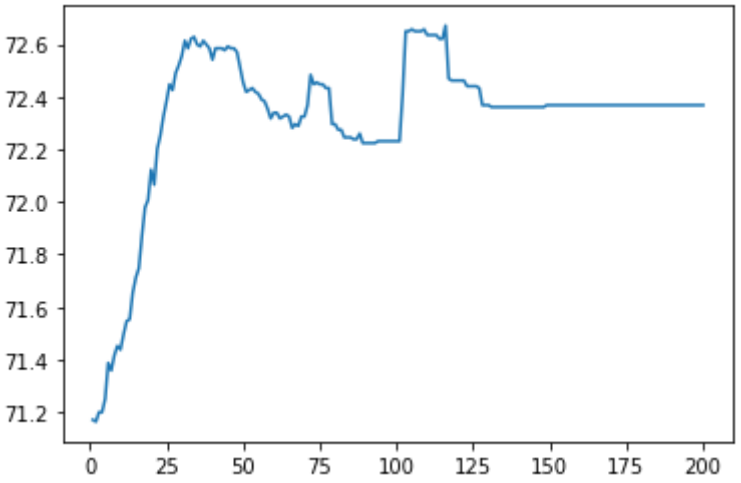
For the final stage of the project we were intending to implement a non-linear SVM. However we've run into some errors and opted to use the linear SVM instead. First of all, we wrote a code that uses a radial basis function. This code is present in the Appendix. The code seems to work when we've inputted a small amount of data. However we've had both Ram and time issues when we've used the entire data set. I've used the Sklearn library to see whether the code we wrote was the problem or not and it had the same problems. We've found that SVM with a non-linear kernel is not efficient to use with big data sets [8], and unfortunately our dataset was too big for our computers or Colab. Creating the feature space requires at least the square of the length of our input set. We've had insufficient RAM errors with our code and couldn't finish the training with Sklearn.

We wanted to implement a kernel approximation method so that we can use a non-linear SVM with our dataset. However, doing this research and trying to identify the problem took a lot of time. At the end we tried RBF-sampling to approximate the kernel using Sklearn and couldn't perceive any meaningful difference in accuracy (at most %73.5 at validation). Since there was no change in accuracy we've opted to use linear SVM. All of these trials are in the Appendix.

Linear SVM:
We have used soft margin linear SVM. We've used loss function can be given as:

$$\lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i - b)\right)\right]$$

Model is fitted using loss function via gradient descent. Its iteration count is found using the validation set.

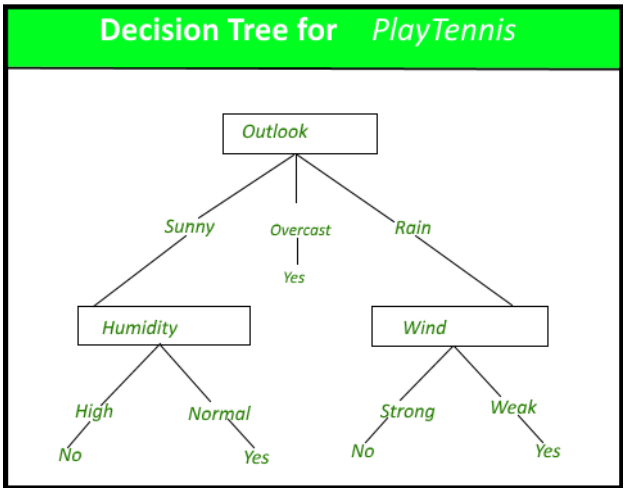


Validation accuracy vs iterations.

*Maximum validation accuracy %72.67 and its iteration is 115. Test accuracy at 115 is %71.25.

3. Decision Tree

Decision Tree is an algorithm which splits the dataset into smaller subsets using the features to pursue a performance metric such as entropy or gini [9]. In our case we considered Gini, since it is supposed to be Computationally faster because Entropy uses logarithm [10]. The branching (i.e., splitting) of the Decision Tree continues until subsets converge to purity by means of Gini, or more practically the predefined maximum depth value is reached. Below, we see an example of a Decision Tree for playing Tennis. It is constructed using the weather information: Humidity and Wind. With each split data is converging into class purity and being separated more accurately.



Similar to this example, our cardio data is being splitted using our features with a greedy search. Which means, our tree is trying to find the best split that will decrease the impurity (Gini index is being exploited for this) greatest. That is the logic behind choosing the root node, and continuing the branching of the tree. Using the features to find the differences between two classes that are having or not having cardiovascular disease, our tree converged to an accuracy of 66.5% with the hyperparameter maximum depth 5 and minimum size 10 (nodes). Moreover, to reduce the bias and variance we used 5 fold cross validation to train the Decision Tree which improved our result. However, using multiple nested loops in the implementation of the tree, resulted in everlasting training time. Hence, for seeing the algorithm's performance we generally had to work with a subset of samples around 1000 instances. Inherent to its nature cross validation is computationally costly considering we have 69,306 data instances (after cleaning), besides, our tree algorithm is not the most efficient one.

Since the computational complexity arose from using multiple for loops to access the data in the decision tree algorithm, cross validation was not feasible for determining the hyperparameters such as maximum depth of the tree. Thus, with research and educated guesses, we decided on using the current values. Moreover, our intention to build a random forest for the final could not be achieved due to training time that takes for even a single tree. Apart from the inefficient implementation of the tree, the existence of 69,306 instances in our dataset is another setback for training time. Therefore, we divided our dataset into 5 folds and calculated the mean accuracy of these folds to come up with the estimate for the accuracy of the tree which was 66.5%. Finally, the poor accuracy compared with other algorithms enabled us to focus on those methods instead of directing our limited time and energy for building a random forest.

Results & Discussion on the Performance

Comparing and contrasting the algorithms we implemented Logistic Regression, SVM and Decision Tree, we should define our metrics to measure the performance of each method.

Accuracy was our main driver while validation of our hyperparameters for the models, therefore, we are mainly using accuracy for the evaluation of our models. Furthermore, execution time should be included in the evaluation, since we encountered some problems about the Decision Tree and its cross validation as it mentioned before. The accuracy values were, 72.5%, 69%, and 68% for Logistic Regression, SVM and Decision Tree, respectively. Training times for Logistic Regression and SVM were not significant unlike the Decision Tree. Logistic Regression was both easy to run and had the highest accuracy. Therefore, we've concluded that out of the methods we've chosen to investigate in this project, the Logistic Regression was the most effective.

References

[1] "Cardiovascular diseases (cvds)," *World Health Organization*, 11-Jun-2021. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/cardiovascular-diseases-(cvds). [Accessed: 28-Feb-2022].

[2] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning with applications in R*, 2nd ed. New York: Springer, 2021.

[3] S. Ulianova, "Cardiovascular disease dataset," *Kaggle*, 20-Jan-2019. [Online]. Available: https://www.kaggle.com/sulianova/cardiovascular-disease-dataset. [Accessed: 28-Feb-2022].

[4] "Low blood pressure (hypotension)," *Mayo Clinic*, 22-Sep-2020. [Online]. Available: https://www.mayoclinic.org/diseases-conditions/low-blood-pressure/symptoms-causes/syc-20355465. [Accessed: 09-Apr-2022].

[5] "Understanding blood pressure readings," *www.heart.org*, 05-Jan-2022. [Online]. Available: https://www.heart.org/en/health-topics/high-blood-pressure/understanding-blood-pressure-readings. [Accessed: 09-Apr-2022].

[6] D. G. Kleinbaum and M. Klein, "Logistic regression," *Statistics for Biology and Health*, 2010.

[7] "Support Vector Machines," *Information Science and Statistics*, 2008.

[8] J. Cervantes, X. Li, W. Yu, and K. Li, "Support Vector Machine classification for large data sets via minimum enclosing ball clustering," *Neurocomputing*, vol. 71, no. 4-6, pp. 611–619, 2008.

[9] Decision Trees. (n.d.). Retrieved April 08, 2022, from https://www.sciencedirect.com/topics/computer-science/decision-trees

[10] P. Aznar, “Decision trees: Gini vs entropy ⋆ Quantdare,” *Quantdare*, 02-Dec-2020. [Online]. Available: https://quantdare.com/decision-trees-gini-vs-entropy/. [Accessed: 09-Apr-2022].

Appendix:

```
import os
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from random import randrange
from random import seed
from google.colab import drive

def summary(data):
    # Looking first few rows of the data
    print(data.head())

    # Looking for the data and missing values
    print(data.info())

    # Looking the summary of the data
    print(data.describe())
    return None

def cleaner(data):
    data = data.iloc[:, 1:] # Deselecting first column
    data.index.name = 'id' # Renaming index of the first column
    data.dropna(inplace = True) # Removing null values
    data['age'] = data['age']/365 # Transforming age into years
    data = data[(data.height>=140) & (data.height<=220)] # Filtering height to be at least 140cm or at most 220cm
    data = data[(data.ap_hi>=50) & (data.ap_hi<=1000)] # Filtering systolic pressure to be at least 50mmHg or at most 300mmHg
    data = data[(data.ap_lo>=20) & (data.ap_lo<=1000)] # Filtering diastolic pressure to be at least 20mmHg or at most 300mmHg
    data = data[data.weight>=40] # Filtering weight to be at least 40
```

```
    # Power transform the data

    return data

def visualizer(data):

    # Scatter plot with cardio against height
    # plt.scatter(data['cardio'], data['height'], c=data['gender'])

    # Adding Title to the Plot
    # plt.title("Scatter Plot")

    # Setting the X and Y labels
    # plt.xlabel('cardio')
    # plt.ylabel('height')

    # plt.colorbar()

    # plt.show()

    pd.crosstab(data['cholesterol'], data['cardio']).plot.bar(stacked=False)
    plt.figure(1)
    pd.crosstab(data['active'], data['cardio']).plot.bar(stacked=False)
    plt.figure(2)
    pd.crosstab(data['gluc'], data['cardio']).plot.bar(stacked=False)
    plt.figure(3)
    pd.crosstab(data['smoke'], data['cardio']).plot.bar(stacked=False)
    plt.figure(4)
    pd.crosstab(data['alco'], data['cardio']).plot.bar(stacked=False)
    plt.figure(5)
    pd.crosstab(data['gender'], data['cardio']).plot.bar(stacked=False)
    plt.show()

    return None

def splitter(seed, data, train = 0.60, val= 0.20):
    if not isinstance(data, (np.ndarray)):
      data = data.to_numpy()

    np.random.seed(seed)
    np.random.shuffle(data)

    indexOfTrain = int(len(data)*train)
    indexOfVal = int(len(data)*(train+val))

    train = data[:indexOfTrain]
    val = data[indexOfTrain:indexOfVal]
    test = data[indexOfVal:]

    return train, val, test

def normalization(datas): # takes numpy array
```

```
  stds = np.std(datas, axis=0)
  means = np.mean(datas,axis=0)
  for i in range(11):
    datas[:,i] = (datas[:,i] - means[i])/stds[i]
  return datas
# pd.set_option('display.max_rows', None, 'display.max_columns', None) # Viewing the output without
truncation
data = pd.read_csv("cardio_train.csv", header= 0, sep= ";")

def label_design(data): # takes numpy array
  labels = data[:, 11]
  datas = np.delete(data, 11, 1)
  return datas, labels

# Looking at the data
summary(data)

# Cleaning the data
data = cleaner(data)

# Looking once again
summary(data) # Now data looks better

# Visualisation of the data
visualizer(data)

# Splitting data
train, val, test= splitter(1, data)
# print(np.shape(train))
# print(np.shape(test))

#Logistic Regression

class lr:
  def __init__(self, learning_rate=1e-4, threshold = 0.5):
    self.learning_rate = learning_rate
    self.threshold = threshold
    self.w = None
  def initialize_weight(self,dim):
    self.w = np.zeros((dim,))
    return None

  def sigmoid(self, x):
    a = np.exp(np.dot(x,self.w))
    return a/(1 + a)

  def fit(self, x_train, y_train):
    dim = x_train.shape[0]
    i = 0
    pre_loss = 0
    loss = 1
    while abs(loss - pre_loss) > 0.01:
```

```
        i += 1
        pre_loss = loss
        gradient = (np.dot(x_train.T, (self.sigmoid(x_train) - y_train)))/dim
        self.w = self.w - gradient*self.learning_rate
        acc = self.accuracy(y_train, self.sigmoid(x_train) > 0.5 )
        y_prob = self.sigmoid(x_train)
        loss = np.sum(-y_train*np.log(y_prob)-(1-y_train)*np.log(y_prob))
        #print(np.linalg.norm(gradient))
        print(i)
        print("Loss: ",loss)
        print("Accuracy: ",acc)

    def test(self, x_test):
      return self.sigmoid(x_test) > self.threshold

    def accuracy(self,y_test,y_est):
      return np.sum(y_est == y_test)/np.size(y_test)

x_train, y_train = label_design(normalization(train))
x_val, y_val = label_design(normalization(val))
x_test, y_test = label_design(normalization(test))
model = lr(1,0.5)
model.initialize_weight(11)
model.fit(x_train, y_train)


accs = []
for i in range(100):
  model.threshold = 0.4 + i/500
  y_est = model.test(x_val)
  accs.append(np.sum(y_est == y_val)/np.size(y_val))
plt.plot(np.linspace(0.4, 0.6, num=100),accs)
plt.ylabel('accuracy')
plt.xlabel('threshold')
plt.show()

index_max = np.argmax(accs, axis=0)
print("Threshold: ", 0.4 + index_max/500)
print("Accuracy: ",np.max(accs))

model.threshold = 0.5
y_result = model.test(x_test)
print("Test Accuracy: ",model.threshold,model.accuracy(y_test,y_result))
model.threshold = 0.45
y_result = model.test(x_test)
print("Test Accuracy: ",model.threshold,model.accuracy(y_test,y_result))

#SVM - Linear

class svm:
    # Linear Soft Margin SVM
```

```
    def __init__(self):
        self.w = None

    def fit(self, x_train, y_train, x_val, y_val, learning_rate=1e-5, epoch=30, regularization_=0.1):
        val_hist = []
        w_list = []
        y_ = (y_train-0.5)*2
        self.w = np.zeros(x_train.shape[1])

        for i in range(epoch):
            print(i,"Train Accuracy: ",self.accuracy(y_train,self.test(x_train)))
            for i in range(len(x_train)):
                if (y_[i] * (np.dot(x_train[i], self.w.T))) >= 1:
                    self.w = self.w - learning_rate * (2 * regularization_ * self.w)
                else:
                    self.w = self.w - learning_rate * (2 * regularization_ * self.w - np.dot(x_train[i], y_[i]))
            val_hist.append(self.accuracy(y_val,self.test(x_val)))
            w_list.append(self.w)
        return val_hist

    def test(self, x_test):
      test_result = []
      for i in range(len(x_test)):
        test_result.append(np.dot(x_test[i], self.w))

      return np.sign(np.array(test_result))

    def accuracy(self,y_test,y_est):
      return 100*np.sum(y_est == y_test)/np.size(y_test)

x_train, y_train = label_design(normalization(train))
x_val, y_val = label_design(normalization(val))
x_test, y_test = label_design(normalization(test))
model = svm()
hist = model.fit(x_train, (y_train-0.5)*2, x_val, (y_val-0.5)*2, learning_rate=1e-6, epoch=200,
regularization_=0.01)

plt.plot(np.arange(1,len(hist)+1),hist)
plt.show()

epoch_max = np.argmax(hist, axis=0)
print("epoch: ", epoch_max)
print("Accuracy: ",np.max(hist))

model_fin = svm()
hist_fin = model_fin.fit(x_train, (y_train-0.5)*2, x_test, (y_test-0.5)*2, learning_rate=1e-6, epoch=epoch_max,
regularization_=0.01)
print("Test Accuracy: ",hist_fin[-1])

#SVM RBF-sampler with sklearn

from sklearn import linear_model, svm, discriminant_analysis, metrics
```

```
from scipy import optimize
x_train, y_train = label_design(normalization(train))
x_val, y_val = label_design(normalization(val))
x_test, y_test = label_design(normalization(test))

model = svm.SVC(kernel='rbf', C=10, gamma=1.5, shrinking=False)
model.fit(x_train, y_train);

from sklearn.kernel_approximation import RBFSampler
from sklearn.linear_model import SGDClassifier
x_train, y_train = label_design(normalization(train))
x_val, y_val = label_design(normalization(val))
x_test, y_test = label_design(normalization(test))
for i in np.arange(100, 1001, 100):
  rbf_feature = RBFSampler(gamma=0.1, n_components=i, random_state=1)
  X_features = rbf_feature.fit_transform(x_train)
  clf = SGDClassifier(max_iter=1000)
  clf.fit(X_features, y_train)
  xt_features = rbf_feature.fit_transform(x_val)
  y_est = clf.predict(xt_features)
  accuracy = np.sum(y_est == y_val)/np.size(y_val)
  print(i," :",accuracy)
#print(accuracy)

xt_features = rbf_feature.fit_transform(x_test)
y_est = clf.predict(xt_features)
accuracy = np.sum(y_est == y_test)/np.size(y_test)
print(accuracy)

#Decision Tree

def gini_index(groups, classes):

    numOfInstances = float(sum([len(group) for group in groups]))
    gini = 0.0

    for group in groups:
        size = float(len(group))

        if size == 0:
            continue
        score = 0.0

        for class_value in classes:
            p = [row[-1] for row in group].count(class_value) / size
            score += p**2

        gini += (1.0 - score) * (size/numOfInstances)

    return gini

def feature_split(index, threshold, dataset):
```

```
    below, above = list(), list()

    for row in dataset:
        if row[index] < threshold:
            below.append(row)
        else:
            above.append(row) # Greater than or equal to values

    return below, above

def find_split(dataset):
            classes = list(set(row[-1] for row in dataset))

            best_col, best_value, best_score, best_groups = 999999, 999999, 999999, None
            for col in range(len(dataset[0])-1):
                        for row in dataset:
                                    groups = feature_split(col, row[col], dataset)
                                    gini = gini_index(groups, classes)

                                    if gini < best_score:
                                                best_col, best_value, best_score, best_groups = col, row[col], gini, groups

            return {'index':best_col, 'value':best_value, 'groups':best_groups}

def terminal(subset):
            result = [row[-1] for row in subset]
            return max(set(result), key = result.count)

def child_split(node, maxDepth, minSize, depth):
    below, above = node['groups']
    del(node['groups'])

    if not below or not above:
        node['below'] = node['above'] = terminal(above + below)
        return
    if depth >= maxDepth:
        node['below'], node['above'] = terminal(below), terminal(above)

    if len(below) <= minSize:
        node['below'] = terminal(below)
    else:
        node['below'] = find_split(below)
        child_split(node['below'], maxDepth, minSize, depth+1)

    if len(above) <= minSize:
        node['above'] = terminal(above)
    else:
        node['above'] = find_split(above)
        child_split(node['above'], maxDepth, minSize, depth+1)

def decision_Tree(train, maxDepth, minSize): # Random Forest might be implemented in the final because of
bias and overfitting possibilities
```

```
    rootNode = find_split(train)
    child_split(rootNode, maxDepth, minSize, 1)
    return rootNode

def treeShow(node, depth = 0):
    if isinstance(node, dict):
        print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value'])))
        treeShow(node['below'], depth+1)
        treeShow(node['above'], depth+1)
    else:
        print('%s[%s]' % ((depth*' ', node)))

def treePredict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['below'], dict):
            return treePredict(node['below'], row)
        else:
            return node['below']
    else:
        if isinstance(node['above'], dict):
            return treePredict(node['above'], row)
        else:
            return node['above']

def main_tree(train, test, max_depth, min_size):
            tree = decision_Tree(train, max_depth, min_size)
            predictions = list()
            for row in test:
                        prediction = treePredict(tree, row)
                        predictions.append(prediction)
            return(predictions)

def crossSplitter(data, numFold):
            dataset_split = list()
            dataset_copy = list(data)
            fold_size = int(len(data) / numFold)
            for i in range(numFold):
                        fold = list()
                        while len(fold) < fold_size:
                                    index = randrange(len(dataset_copy))
                                    fold.append(dataset_copy.pop(index))
                        dataset_split.append(fold)
            return dataset_split

def accuracyCalc(ground, prediction):
            trueCount = 0
            for i in range(len(ground)):
                        if ground[i] == prediction[i]:
                                    trueCount += 1
            return  100.0 * trueCount / float(len(ground))

def accuracyChecker(dataset, classifier, numFolds, *args):
```

```
        folds = crossSplitter(dataset, numFolds)
        accuracyValues = list()
        for j in folds:
                trainingSet = list(folds)
                trainingSet.remove(j)
                trainingSet = sum(trainingSet, [])
                testingSet = list()
                for row in j:
                        row_copy = list(row)
                        testingSet.append(row_copy)
                        row_copy[-1] = None
                prediction = classifier(trainingSet, testingSet, *args)
                ground = [row[-1] for row in j]
                accuracy = accuracyCalc(ground, prediction)
                accuracyValues.append(accuracy)
        return accuracyValues


seed(5)

os.chdir('C:/Users/senih/Desktop/Coding/VS/Python/ee485')
dataset = pd.read_csv("cardio_train.csv", header= 0, sep= ";")

dataset = dataset.to_numpy()

dataset = dataset[:1000, :] # We will focus on efficiency of the algorithm to make it faster
# print(np.shape(dataset))

dataset = dataset.tolist()

numFolds = 5
maxDepth = 5
minSize = 10

accuracyValues = accuracyChecker(dataset, main_tree, numFolds, maxDepth, minSize)
print('accuracyValues: %s' % accuracyValues)
print('Mean Accuracy: %.3f%%' % (sum(accuracyValues)/float(len(accuracyValues))))

#SVM - Kernel (Not in use)

import numpy as np
import cvxopt
import cvxopt.solvers

class SVM():
  def __init__(self,polyconst=1,gamma=10,degree=2):
    self.polyconst = float(1)
    self.gamma = float(gamma)
    self.degree = degree
    self._support_vectors = None
    self._alphas = None
    self.intercept = None
```

```
    self._n_support = None
    self._support_labels = None
    self._indices = None

  def transform(self,X):
    K = np.zeros([X.shape[0],X.shape[0]])
    for i in range(X.shape[0]):
      print(i)
      for j in range(X.shape[0]):
        K[i,j] = np.exp(-1.0*self.gamma*np.dot(np.subtract(X[i],X[j]).T,np.subtract(X[i],X[j])))
    return K

  def fit(self,data,labels):
    num_data, num_features = data.shape
    labels = labels.astype(np.double)
    alphas = np.ravel(cvxopt.solvers.qp(cvxopt.matrix(np.outer(labels,labels)*self.transform(data)),
                                        cvxopt.matrix(np.ones(num_data)*-1),
                                        cvxopt.matrix(labels,(1,num_data)),
                                        cvxopt.matrix(0.0),
                                        cvxopt.matrix(np.diag(np.ones(num_data) * -1)),
                                        cvxopt.matrix(np.zeros(num_data)))['x'])
    is_sv = alphas>1e-5
    self._support_vectors = data[is_sv]
    self._n_support = np.sum(is_sv)
    self._alphas = alphas[is_sv]
    self._support_labels = labels[is_sv]
    self._indices = np.arange(num_data)[is_sv]
    self.intercept = 0
    for i in range(self._alphas.shape[0]):
      self.intercept += self._support_labels[i]
      self.intercept -= np.sum(self._alphas*self._support_labels*K[self._indices[i],is_sv])
    self.intercept /= self._alphas.shape[0]

  def signum(self,X):
    return np.where(X>0,1,-1)

  def project(self,X):
    score = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
      s = 0
      for alpha,label,sv in zip(self._alphas,self._support_labels,self._support_vectors):
        s += alpha*label*np.exp(-1.0*self.gamma*np.dot(np.subtract(X[i],sv).T,np.subtract(X[i],sv)))
      score[i] = s
    score = score + self.intercept
    return score

  def predict(self,X):
    return self.signum(self.project(X))

set1, data1 = splitter(1, data, 0.01)
set2, data2 = splitter(1, data1, 0.01)
x_train, y_train = label_design(normalization(set1))
```

```
x_test, y_test = label_design(normalization(set2))
model = SVM(gamma=3)
model.fit(x_train,y_train)

predictions = model.predict(x_test)

y_guess = model.predict(x_test)
control_param = 0
result = np.sum(y_guess == y_test)/np.size(y_test)
print(result)
```