Atakan Topcu - 21803095
Ata Yavuzyılmaz - 21802466 EEE485-02

EEE 485 Term Project Final Report - Heart Disease Predictor

Introduction:

Heart diseases are the number 1 cause of death globally, with around 17.9 million lives dying each year [1]. Thus, in order to understand how each risk factor affects the possibility of having heart disease and create a predictive model that can give us an idea about the relationship between risk factors and heart diseases, our objective of this project has formed to be implementing a heart disease predictor. We have planned to use 3 different learning algorithms that are SVM, kNN, and Shallow Neural Network. Our dataset can be found from kaggle.com and it is called "Heart Disease UCI" [2]. With these algorithms, the accuracy of each method will be investigated and compared with the others.

Attributes Description:

The diagnosis of heart disease is done on a combination of clinical signs and test results. It is found that age, gender, family history, smoking, blood pressure, cholesterol, diabetes, obesity, and stress levels are the factors for diagnosing heart disease [3]. The dataset also contains other variables such as the number of major vessels (lesser vessels would mean a higher chance of heart disease when a coronary artery is blocked) and some possible symptoms such as chest pain type and exercise-induced angina. These attributes will be explained more clearly below. It is as follows.

1. **Age:** Age is a continuous value that is represented in years. Older patients are more vulnerable to heart disease because of their damaged arteries and weakened heart muscles [3].
2. **Sex:** Males have a higher probability of having heart diseases. This is a categorical value that is equal to 1 when the patient is male and 0 when the patient is female [3].
3. **Chest Pain Type (cp):** This categorical attribute is equal to 0 when the patient is experiencing typical angina, which is a result of oxygen deficiency at the heart [4]. It is 1 for atypical angina and 2 for non-cardiac chest pain. Both of these chest pain types are not related to the heart. The attribute is equal to 3 when the patient does not experience any chest pain.
4. **Resting Blood Pressure (trestbps):** A continuous value in mmHg. Blood pressure is an important risk factor for heart diseases because the pressure can damage the arteries [3].
5. **Serum Cholesterol (chol):** A continuous attribute in mg/dl. High blood cholesterol can cause atherosclerosis, which means plaque formation inside arteries that hardens and narrows them, resulting in lower blood circulation to the heart [5].
6. **Fasting Blood Sugar (fbs):** Diabetes increases the risk of heart disease [3]. If the fasting blood sugar is higher than 126 mg/dl, it means that the patient has diabetes [6]. This attribute is 1 if fasting blood sugar is higher than 120 mg/dl, and 0 otherwise.
7. **Resting Electrocardiographic Results (restecg):** It is 0 when there are no problems seen at the ECG results, 1 if the patient has ST-T wave abnormality, which may be a symptom or incidental [7]. It is 2 if the patient has left ventricular hypertrophy, meaning thicker walls at the left chamber of the heart which results in inefficient pumping [8].
8. **Maximum Heart Rate (thalach):** A continuous attribute in bpm.
9. **Exercise-Induced Angina (exang):** 1 if the patient experiences angina, heart-related chest pain, after exercise. 0 otherwise.
10. **ST Depression Induced by Exercise Relative to Rest (oldpeak):** It is a continuous variable in mm. An ST depression higher than 1 mm can be a result of ischemia [9], which means low blood supply to the body [10].
11. **The slope of the Peak Exercise ST-Segment (slope):** The ST segment slope induced by exercise, is upwards if the value is 0, flat if the value is 1, and downwards if it is 2.
12. **The number of Major Vessels (ca):** If one of the vessels is blocked, the blood transportation can be achieved through other vessels. So, more major vessels mean less chance of heart disease.
13. **Thallium Stress Result (thal):** Thallium stress test is conducted to see whether the blood is flowing well to the heart or not [11]. It is 1 if the results are normal, 2 if there was a defect that is fixed, and 3 if there is a reversible defect.
14. **Have Disease or Not (target):** 1 if the patient has heart disease, 0 otherwise.

Review of Machine Learning Algorithms:

1. **K-nearest Neighbours (kNN):** This is a supervised machine learning algorithm since it requires labeled input data to learn a function that produces an appropriate output [12]. Its logic is fairly simple. The basic principle is that we select a vector from the dataset and we compare its distances to other vectors. We take the k of these smallest distances and then, depending on the majority of these selected vectors' output, chosen vector's output is decided by a majority of the output (Votes) [13] [14]. Therewithal, it does not require any training as it merely calculates the distance between vectors. As we can see, the algorithm itself is fairly simple. However, it still has some setbacks. One of the main disadvantages of the algorithm is that it can be computationally expansive since it first takes the Euclid norm of every vector with the chosen vector. So, if we increase the size of the dataset, the speed of the algorithm decreases which affects the efficiency of the method. Another problem is that we need feature scaling for this method. This is because the algorithm uses Euclidean distance for predicting the class so if the feature vector is not scaled distance will be dominated by the feature that has the broad range. Thus, normalization is necessary. Furthermore, one of the most critical problems of kNN is that there is no optimal way of choosing the value of k. This problem was also present in our case which we will discuss more when we talk about the accuracy of the methods. Since there is no definite way of choosing k, we have to use some heuristics such as the elbow method to tackle the problem which does not solve the problem itself [15]. However, even with heuristics, it might still lead to faulty conclusions which is why it is called heuristics. Moreover, another issue of the algorithm is that it has no capability of dealing with missing values. However, since our dataset does not have missing problems, it did not create any hurdle for us.

2. **Shallow Neural Network (SNN):** Shallow neural networks are simply neural networks with one hidden layer [16]. It consists of perceptions that are interconnected with each other. In the input layer, each perceptron takes inputs with their weight along with their bias. Then, it computes the output by these components' summation. After that, it performs activation to give the final output [16]. However, activation functions can be changed depending on the given dataset and the problem such as ReLU, Softplus, or Logistic function. Then each of these output values is used as an input to the perceptron of the hidden layer. It can be formulized as follows,

$$\boldsymbol{v}^{(l)} = \sum_{\boldsymbol{i=0}}^{\boldsymbol{d(l-1)}} \boldsymbol{w}_{\boldsymbol{ij}}^{(\boldsymbol{l})} \boldsymbol{x}_{\boldsymbol{i}}^{(\boldsymbol{l-1})} \; where\ l\ denotes\ the\ layer,$$

$$d(l)\ denotes\ the\ number\ of\ neurons\ in\ layer\ l\ and\ j\ denotes\ the\ neuron\ in\ the\ current\ layer$$

$$\boldsymbol{x}_{\boldsymbol{j}}^{(\boldsymbol{l})} = \boldsymbol{Activation\_Function}(\boldsymbol{v}_{\boldsymbol{j}}^{(\boldsymbol{l})})$$

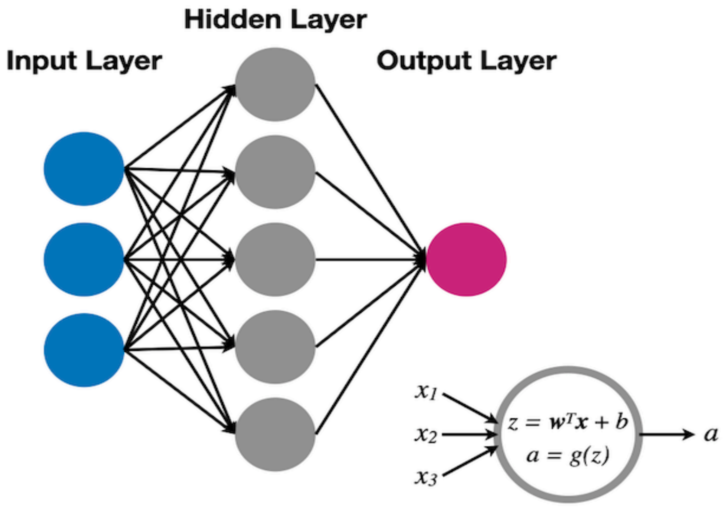


Figure 1. Shallow Neural Network Demonstration [17].

Weights of the neural network are randomly initiated [16]. The neural network is then trained by repeated iterations of backpropagation [18]. The input is given to the network, and the loss function is computed based on the output. After that, all the weights of the connections between the layers are updated to minimize the loss function until we reach convergence. The loss function for a single vector in the dataset is as follows,

$$l(f_w(x_i), y_i) = \left(y_i - f_w(x_i)\right)^2 \; where \; x_i = \left[x_{i,0}\; x_{i,1}\; x_{i,2} \ldots x_{i,d}\right]^T$$

The summation of these error functions for all samples of the dataset yields the loss function of the whole dataset. After that, we update the weights by using Stochastic gradient descent which applies the gradient descent algorithm to randomly selected samples of the dataset since using the whole dataset would be computationally expensive. Thus SGD shortens the time to train.

$$w_{ij}{}^{(l)} = w_{ij}{}^{(l)} - \gamma \frac{\partial l(f_w(x_i), y_i)}{\partial w_{ij}^{(l)}} \; where \; \gamma = learning\; rate$$

Furthermore, since the update of the weights continues until they converge, neural networks are more prone to overfitting the train dataset. To avoid that we need to apply weight decay for the backpropagation. The following equation gives the final version of the SGD algorithm.

$$w_{ij}{}^{(l)} = (1 - \gamma\lambda) w_{ij}{}^{(l)} - \gamma \frac{\partial l(f_w(x_i), y_i)}{\partial w_{ij}^{(l)}}$$
$$where \; \gamma = learning\; rate, \lambda = regularization\; parameter$$

Using weight decay decreases our accuracy on the training dataset while it increases the accuracy on the test dataset. We can conclude that the neural network algorithm is very flexible, thus it can easily deal with complex problems. However, its algorithm requires tweaking and guesswork just like in kNN. In SNN's case, we need to decide on the activation algorithm according to our dataset, and also for weight decay, there is no optimal way to find the regularization parameter. Just like in kNN, we can simulate the model for different regularization parameters and choose the best one. However, this would also take time. Furthermore, apart from the regularization parameter, the model itself can take a lot of time to train. Thus, it can be said that there is a trade-off between the performance of the system with the time it takes to train.

3. **Support Vector Machine (SVM):** A support vector machine is an algorithm that divides the data instances with different results in the most optimal way [18]. There are many different ways to divide the data instances that are spread on the n-dimensional plane, however, SVM is the most optimal because it divides with a margin. This margin is maximized and the data is divided in the best way possible [19]. This is called the large margin principle [19]. SVMs are mostly used with kernels to separate the data in nonlinear ways, but in this project, no kernel is used and a linear SVM is implemented to reduce the computational cost and the time to train the algorithm. To implement SVM, we first start with a vector **$\underline{w}$**, which is perpendicular to the margin lines, and take its dot product with a data vector **$\underline{x}_i$** [20]. Since a dot product is simply a projection, it can be understood that if the result of the dot product is larger than a constant b, it is classified as a positive sample [20]. On the contrary, if the result is smaller than b, it is classified as a negative sample. Just like kNN, to eliminate the possibility of a feature with a wide range to dominate the results, the feature vectors should be standardized to get better results. The objective function of SVM is:

$$J = \frac{1}{2}\left|\left|\underline{w}\right|\right|^2 + C\sum_{i=1}^{N}\left(1 - y_i\left(\underline{w}\cdot\underline{x_i} + b\right)\right)$$
$$where \; y_i = 1 \; for\; positive\; samples, -1 \; otherwise\; and\; C = Regularization\; parameter$$

Since the results of our dataset are 1 for positive samples and 0 for negative samples, the data should be processed accordingly before implementing the SVM. The objective function is then minimized with respect to **w**, to find the most optimal weight vector [20]. Just like the SNN algorithm, the objective function can be minimized and **w** can be found using Stochastic Gradient Descent. The gradient of the objective function is calculated and the weights are updated according to the gradient until the weight vector converges.

$$\boldsymbol{\overline{w_{n+1}} = \overline{w_n} + \gamma \Delta J \ where\ \gamma = Learning\ rate}$$

Unlike kNN, SVM requires training to predict the values. Calculating the weight vector of SVM can take some time depending on the regularization parameter, learning rate, and the number of iterations done for gradient descent. Just like SNN and kNN, these parameters require fine-tuning such that the accuracy of the algorithm is maximized. Training SVM takes less time compared to SNN because the weight vector of SNN is larger. However, it is expected that SNN is more accurate than SVM because it can separate nonlinearly separable data better. Here, SVM is used without a kernel, so it can only separate linearly separable data perfectly. If a kernel is used, training time will increase, which means that there is a trade-off between the training time and accuracy, as it is mentioned in the previous part.

Pre-processing of the Dataset:

In the pre-processing of the dataset, we dealt with the feature scaling along with creating dummy variables for the categorical features. Furthermore, we also looked at the correlations of each attribute with each other so that we could determine how they are related to each other. Furthermore, a 3d plot of the continuous variables is also plotted to observe the correlation between them more clearly. These extra plots can be generated using the code given in the Appendix for better resolution as we have to conform the given page limit. The most important figures of the dataset are given in figure 2.

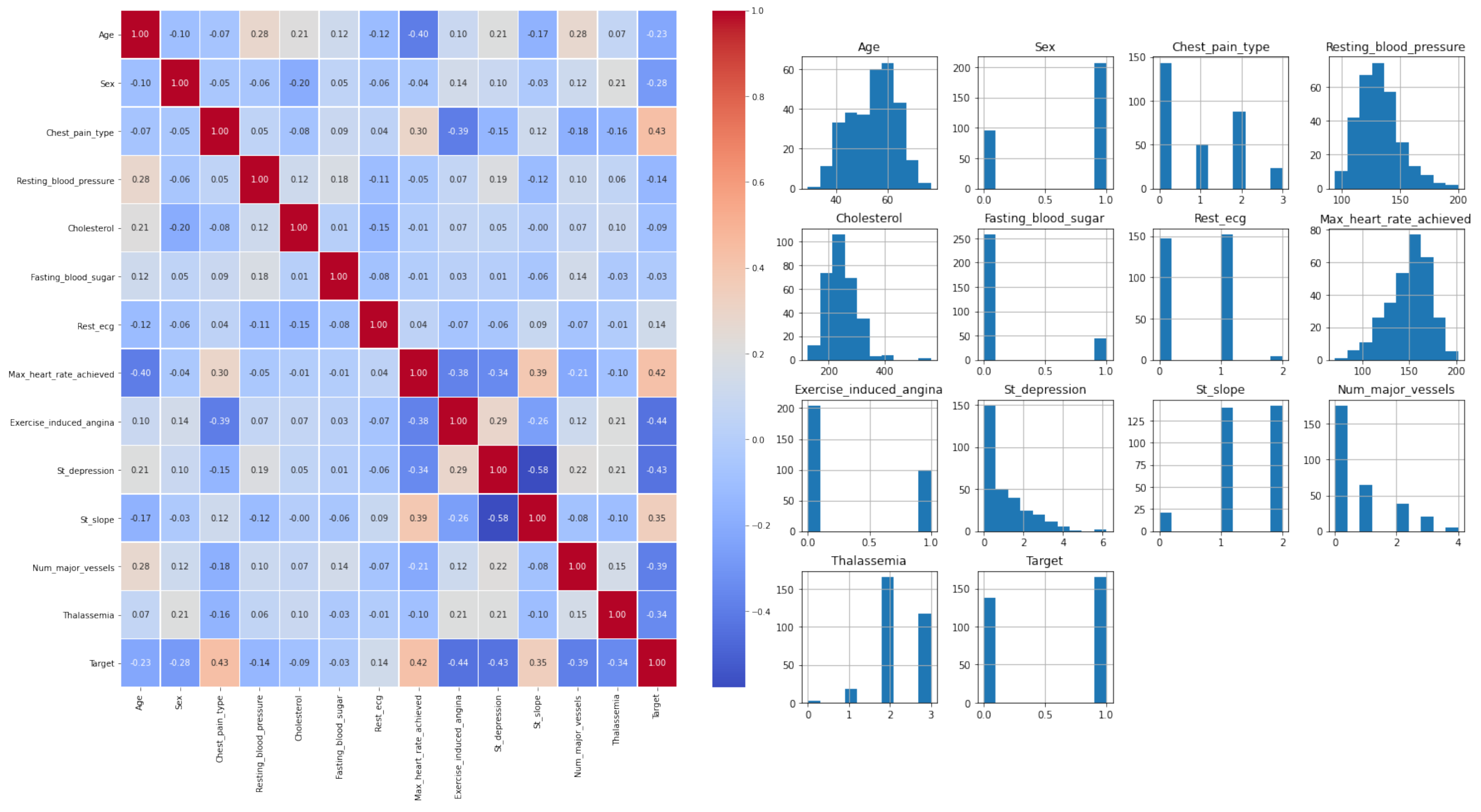


Figure 2. Correlation matrix of each attribute along with their respective histograms.

From investigating the dataset, we have concluded that the size of the original dataset matrix is 303x14. Furthermore, it has been observed that the dataset does not contain any missing data which would hinder our kNN algorithm. However, it has been observed that our output values are not evenly matched. That is the number of people that have heart disease is 165 while the number of healthy people is 138. This means that in algorithms like kNN, it will be more likely to classify an unknown case as heart disease. However, since the gap in numbers is not high, we still got great accuracy in the simulations. Furthermore, since we used k-fold cross validation out so that it reduces the high variance in the accuracy of the dataset. After looking at some statistics of the dataset such as the minimum and maximum of the attributes. We have seen that most of the attributes are categorical. Furthermore, continuous variables have different scales than each other which meant most of our algorithms would not correctly work if we do not normalize these attributes.

- **Categorical Features:** Sex, Chest pain type, Fasting blood sugar, Resting electrocardiographic results, Exercise-induced angina, ST slope, Number of blood vessels, Target.
- **Continuous Features:** Age, Resting blood pressure, Cholesterol, Maximum heart rate, ST depression.

After identifying the categorical values of the dataset, we have used panda's "get_dummies" command to create dummy variables excluding the Target attribute which is our output value.

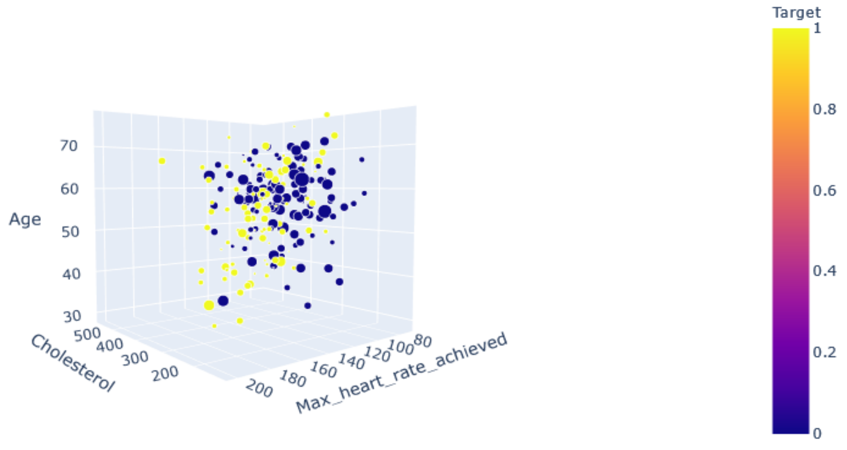
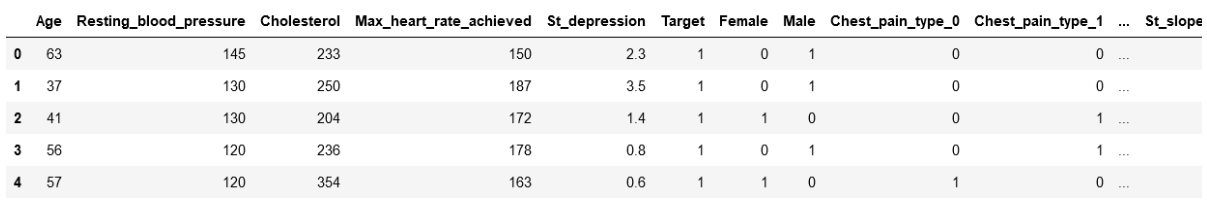


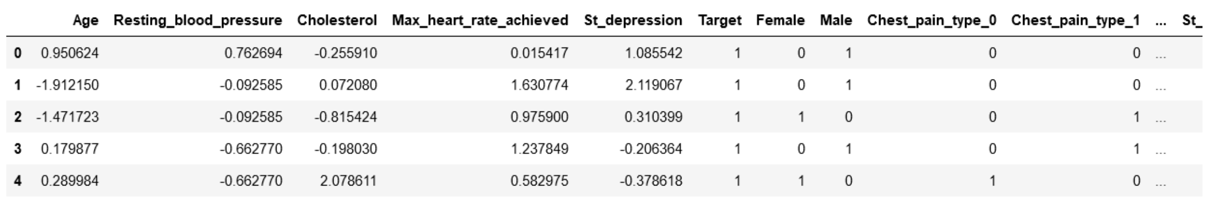
Figure 3. 3d plot of the continuous features where colours represent the target value (Resting Blood Pressure excluded).

After creating the dummy variables, our new dataset's matrix size became 303x31. These dummy variables however can create multicollinearity. Because of that, there may be problems in algorithms such as extra weight on the distance calculation on kNN or it will take more time to converge for neural network weights. Though they would not affect the accuracy of the system as weights of these variables would negate each other, they might take up time due to an increase in the number of variables. Furthermore, since we have already mentioned the need for feature scaling for some algorithms, normalization applied to the feature vector as it would affect the efficiency of kNN and SVM. This normalization only applied to continuous variables as it would be meaningless to apply them to categorical/dummy variables as they can only take up 0 and 1. There were two options that we have thought about feature scaling. One was the min-max scaling while the other one was standardization. Though we can say that min-max scaling can be considered useful for algorithms like kNN as it limits every variable range to be the same for distance measures, it does not handle the outliers very well. On the other hand, standardization easily handles outliers while it does not give the same range for every variable [21]. Therefore, we have chosen standardization to scale our dataset as it is handling outliers better than min-max scaling and hence, reduces the probability of efficiency loss due to outliers.



	Age	Resting_blood_pressure	Cholesterol	Max_heart_rate_achieved	St_depression	Target	Female	Male	Chest_pain_type_0	Chest_pain_type_1	...	St_slope
0	63	145	233	150	2.3	1	0	1	0	0	...	
1	37	130	250	187	3.5	1	0	1	0	0	...	
2	41	130	204	172	1.4	1	1	0	0	1	...	
3	56	120	236	178	0.8	1	0	1	0	1	...	
4	57	120	354	163	0.6	1	1	0	1	0	...	

Figure 4. a. Dataset before feature scaling.



	Age	Resting_blood_pressure	Cholesterol	Max_heart_rate_achieved	St_depression	Target	Female	Male	Chest_pain_type_0	Chest_pain_type_1	...	St_
0	0.950624	0.762694	-0.255910	0.015417	1.085542	1	0	1	0	0	...	
1	-1.912150	-0.092585	0.072080	1.630774	2.119067	1	0	1	0	0	...	
2	-1.471723	-0.092585	-0.815424	0.975900	0.310399	1	1	0	0	1	...	
3	0.179877	-0.662770	-0.198030	1.237849	-0.206364	1	0	1	0	1	...	
4	0.289984	-0.662770	2.078611	0.582975	-0.378618	1	1	0	1	0	...	

Figure 4. b. Dataset after feature scaling.

After the pre-processing of the dataset, we have proceeded with choosing the validation method for algorithms.

Validation of the Algorithms:

Since just randomly splitting the dataset would mean high variance in MSE of these differently selected test sets and thus, overestimation of the test error. We needed to either use Leave-one-out cross-validation or k-fold cross-validation. Though we know that Leave-one-out cross-validation eliminates the randomness of the test errors and thus minimizing the variance and overestimation of the test error, it would be computationally expensive. Thus, we have chosen k-fold cross-validation for reducing the overestimation of the test error while reducing the needed computational resources for it by choosing k moderately (k=6). For the first part of our project, we have used k-fold cross-validation to see our methods' accuracy. After the feedback we got from our TA, we have realized that we have to train and test our algorithms with the same chosen dataset. Otherwise, we cannot compare algorithms with each other. For parameter tuning, we shouldn't use the test dataset. Thus, after the feedback, we have used k-fold cross-validation for determining the optimal algorithm parameter values (C and Hidden neuron number). We have divided the total dataset into two different datasets. For training, we have used 0.7 of the original dataset and the rest is used for the testing phase. Then, the training dataset is divided into 6-folds to perform cross-validation. Without using the training dataset, we have obtained 6 different accuracy scores for each C and hidden neuron number. The parameters that have the highest average accuracy score are used in the algorithm. After determining our parameters, we have used the training dataset to train, and test dataset to test all of them. By doing that, we can compare algorithms with each other since they used the same instances of the dataset.

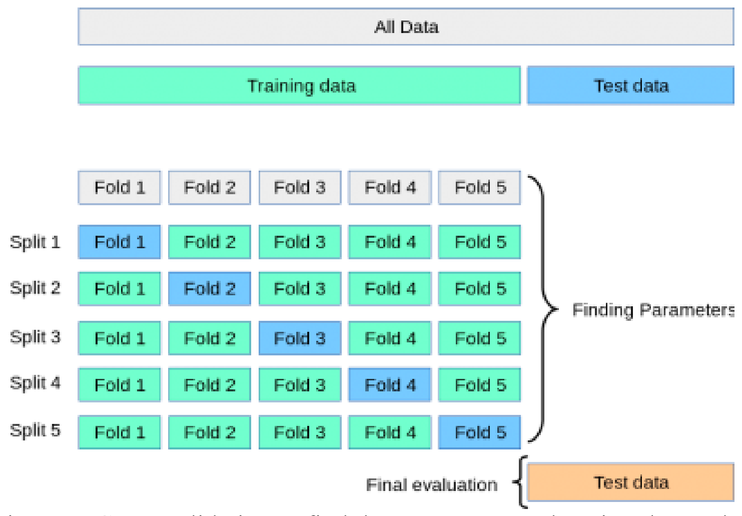


Figure 5. Cross-validation to find the parameters and testing the models.

Simulation and Performance of the Chosen Methods:

As it is already stated, the kNN algorithm doesn't require any training, it just compares the selected data vector with the training dataset. Thus, it does not have separate training and testing phases which means it has a single accuracy. Our dataset has 303 data instances and 31 dimensions after adding dummy variables, therefore it is a relatively small dataset. For comparison purposes, we have used 212 data instances for training. kNN calculates the vector norm of all data instances in the training matrix, since the matrix is small, it is fast. The time it takes to predict the test matrix takes approximately 0.28 seconds. Since we decrease the size of the dataset which was already small, it is expectable. The mean accuracy of the kNN algorithm was around 83-84% in the previous demo. However, with the reduction of the training matrix, the accuracy of the kNN algorithm turns out to be around 79%. We can say that the kNN algorithm can be considered successful if we increase the size of the dataset. Since the training dataset has 212 instances compared to its original size of 303, some information of the data gets lost. Thus, it can be said that it might affect the accuracy of the algorithm due to loss in samples.

Since SVM uses Stochastic Gradient Descent to find the weight vector, it takes more time to train the algorithm. SGD is iterated in a chosen number of epochs so that the weight vector converges and the objective function is minimized. For the first demo, the number of epochs was 5000. The time it takes to train SVM for a single fold was 6.6 seconds. This training time is decreased in the second demo by making the number of epochs 200. Then the time it takes to train for one fold became 0.31 seconds. This time can also be decreased by changing the learning rate of SGD and the regularization parameter of the objective function. In the first demo, the accuracy of train data of SVM was around 88-89%, and the accuracy of test data of SVM was around 83-84%. After we have used the same training matrix that we have used for kNN, our results slightly changed. Currently, the accuracy of train data of SVM is around 86%, and the accuracy of test data of SVM is around 81%. The decrease in accuracy can also happen because the number of epochs is decreased compared to the first demo. Unlike the previous case, we did not use 6-fold cross-validation for accurate measurement. We used that for the determination of the parameter C. For comparison, we have used the only single fold cross-validation since it takes a lot of time to execute the functions for all algorithms six times. Due to a decrease in the fold number, our accuracy measurements are unfortunately susceptible to variance.

The total elapsed time for SNN is higher than SVM since there are more weights at multiple layers. To calculate the weight vector of SNN, back-propagation with weight decay is used, which is a different type of gradient descent [18]. The weight vector converges for every layer in SNN, therefore, the backpropagation is executed twice. After training SNN, the neurons in SNN perform a weighted sum and the activation function is activated. This means that the prediction time of a trained SVM model should be lower than the prediction time of a trained SNN model since SVM only performs a single dot product but SNN performs multiple dot products that are inputted in the chosen activation function. Even though it takes time to train an SNN model, its results are more accurate since it can separate data nonlinearly, as it is mentioned in the Review part. Furthermore, since we used weight decay for the SNN, there is less chance for the algorithm to be overfitted for the training dataset. Therewithal, we have observed that the accuracy of train data of SNN is around 87%, and the accuracy of test data of SNN is around 85%. For the reasons that we have mentioned above, we can see that SNN indeed has better accuracy than SVM.

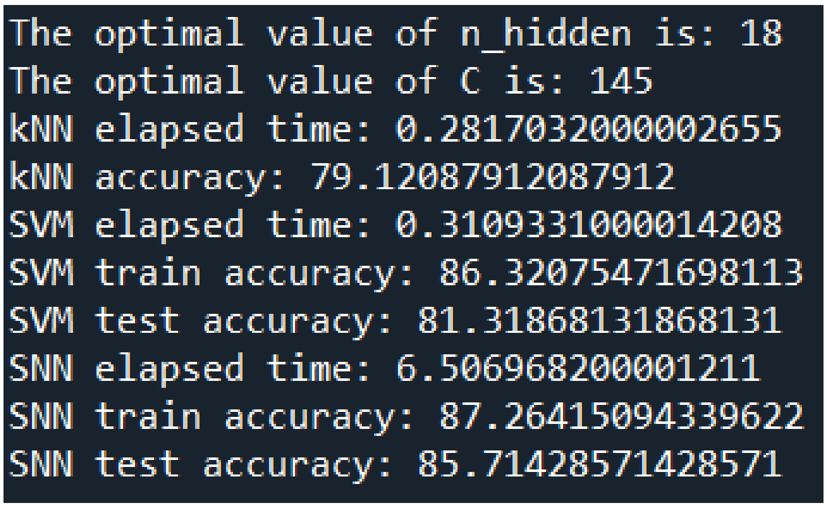

```
The optimal value of n_hidden is: 18
The optimal value of C is: 145
kNN elapsed time: 0.2817032000002655
kNN accuracy: 79.12087912087912
SVM elapsed time: 0.3109331000014208
SVM train accuracy: 86.32075471698113
SVM test accuracy: 81.31868131868131
SNN elapsed time: 6.506968200001211
SNN train accuracy: 87.26415094339622
SNN test accuracy: 85.71428571428571
```

Figure 6. Performance results of the algorithms.

It can be seen from Figure 6 that the accuracy of every algorithm can be considered high, however, the execution time of kNN is small compared to others. kNN doesn't need any training, and the time it takes to train SVM and SNN is higher because of SGD. For the first demo, the epoch number of SVM was 5000 and it was trained in 6.6 seconds. After decreasing it to 200, it became 0.31 seconds, which is very low. The epoch number of SNN is also 200, but as we mentioned before, it is expected that SNN has a larger training time compared to SVM because of its multiple layers. After training the SVM model, the time it takes to predict any given test dataset is very low. This is the disadvantage of kNN. For every test matrix, it requires that the norm is calculated for all train data again, which results in extra computation cost. SVM is trained for a single time, however, after its training, it only computes a dot product. If we have a system that is used very frequently for prediction, a trained SVM model can save a lot of time and since their accuracy is similar to each other we would not be at loss in terms of accuracy. Furthermore, if the data matrix is too large, computing the distance for every data instance may take a lot of time for kNN. For SVM, this will also result in higher training time, but after training, the time for prediction will still be relatively low. In the case of our dataset, we can see that accuracy of SVM is higher than the accuracy of kNN. For SNN, we can see that it has the highest test accuracy. However, it has also the highest execution time. Thus, depending on the goal, If high accuracy is needed, it might be appealing to implement SNN compared to other algorithms. In the case of our dataset, we can see that it has the highest accuracy. Thus, we can consider it to be the best algorithm for our case even though it takes more time for SNN to be trained.

To find the value of k in the kNN algorithm, we have come up with several approaches. The first one was to apply the elbow method which is incrementing the value of k for each iteration and finding the maximum accuracy. However, it was observed that this method did not give a stable result of k. At each attempt, it gave different results. As it was stated before this was one of the main problems of the kNN algorithm. Since there were no definite ways to find the k, we have used the conventional method for finding k which is k=sqrt(N) where N represents the attribute number.

To find the value of C that maximizes the accuracy of SVM, cross-validation is used. The result can be observed in Figure 7. C values are tested between 5 and 150. The optimal value of C is found to be 145. Again, because of the randomness, this value and the plot might change but the change is low compared to the elbow method in kNN so that it is neglected.

To find the number of hidden neurons that maximizes the test accuracy of SNN, cross-validation is used as in the SVM case. The result can be observed in Figure 7. Neuron numbers are chosen between 8 and 25. The optimal number of neurons is found to be 18. Just like in the previous cases, this value and the plot might change but the change is significantly lower compared to the elbow method in kNN. Thus, it can be neglected.

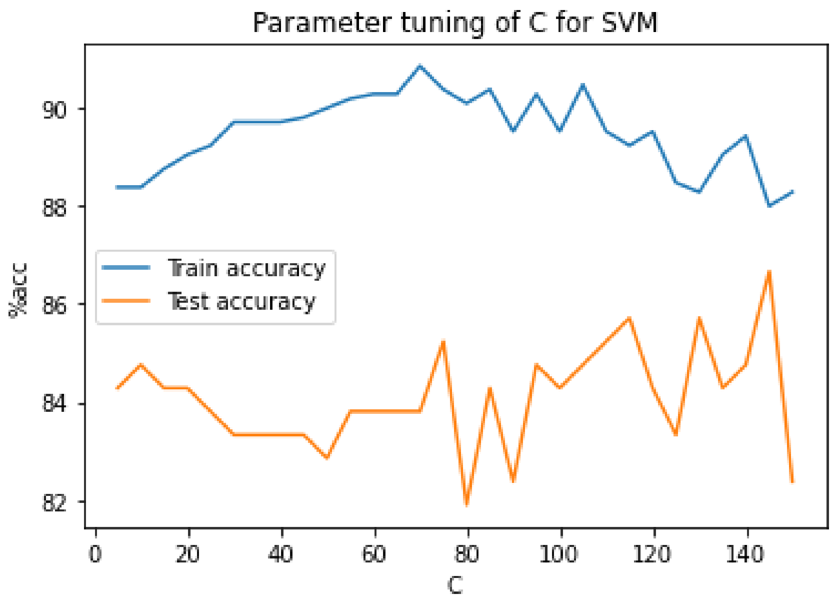


Figure 7. Train accuracy (Blue) and Test accuracy (Yellow) with respect to C.

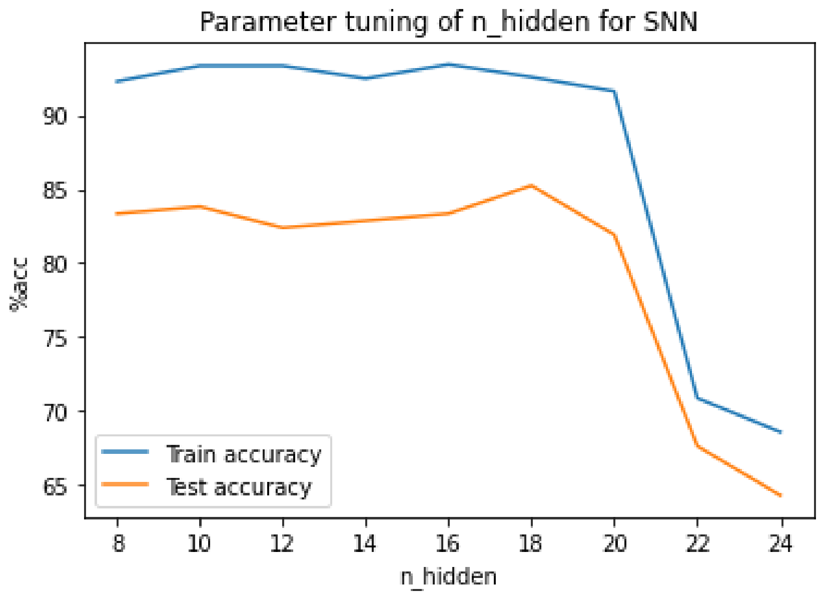


Figure 8. Train accuracy (Blue) and Test accuracy (Yellow) with respect to neuron number (n_hidden).

Conclusion:

Throughout the project, the accuracy of the algorithms (kNN, SVM, and SNN) are investigated for the used dataset. For better accuracy, the dataset was pre-processed first. In this stage, correlations between attributes are observed, necessary graphs were plotted for better visualization and categorical attributes were identified. Furthermore, dummy variables were created for these categorical attributes. After creating the dummy variables, feature scaling was applied to increase the performance of the algorithms. After writing the code for each algorithm, each algorithm was evaluated using k-fold validation. However, since we do not use the same data instances for training and testing, evaluations that we gathered from the first phase did not give enough data for comparison of algorithms. Thus, after the feedback, we have created a new python function and used the same dataset for training and the same dataset for testing for every algorithm. k-fold cross-validation is performed on the training dataset to tune the parameters: the number of hidden neurons for SNN and C for the SVM algorithm. The test dataset shouldn’t be used for cross-validation, therefore, after tuning the parameters the model is trained and tested without splitting the dataset into folds. Since we did not use k-fold for comparison of the algorithms, variance in the accuracies can be expected. Furthermore, dummy variables can lead to multicollinearity. It may lead to extra weight on the distance calculation on kNN or it will take more time to converge for neural network weights. However, this would not affect the accuracy but the elapsed time of the algorithms. Nevertheless, it was observed in the end that SNN has the highest test accuracy among the algorithms. Thus, we can say that for determining a heart disease, SNN would be a better choice than the others even though it requires more time than other algorithms. This was an expected result as neural networks are highly flexible even though they may not be the most efficient algorithm. All in all, the project gave a promising outcome for every algorithm. Among those algorithms, the best algorithm was observed to be a shallow neural network.

Gannt Chart:

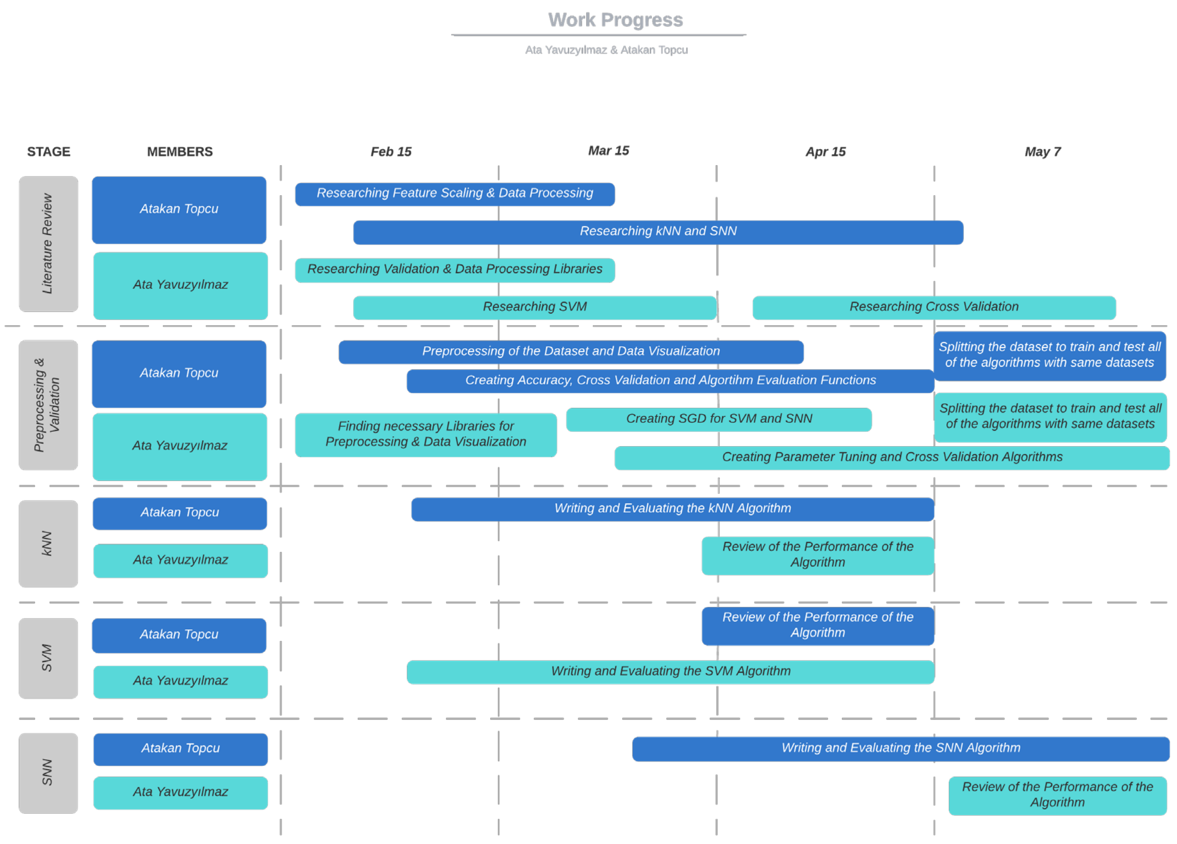


Figure 9. Gannt chart of the project.

References:

[1] https://www.who.int/health-topics/cardiovascular-diseases#tab=tab_1
[2] https://www.kaggle.com/ronitf/heart-disease-uci
[3] https://www.mayoclinic.org/diseases-conditions/heart-disease/symptoms-causes/syc-20353118
[4] https://www.harringtonhospital.org/typical-and-atypical-angina-what-to-look-for/
[5] https://www.nhlbi.nih.gov/health-topics/atherosclerosis#%3A%7E%3Atext%3DAtherosclerosis%20is%20a%20disease%20in%2Cother%20parts%20of%20your%20body
[6] https://www.mayoclinic.org/diseases-conditions/diabetes/diagnosis-treatment/drc-20371451
[7] https://www.healio.com/cardiology/learn-the-heart/blogs/68-causes-of-t-wave-st-segment-abnormalities
[8] https://www.mayoclinic.org/diseases-conditions/left-ventricular-hypertrophy/symptoms-causes/syc-20374314
[9] https://en.wikipedia.org/wiki/ST_depression
[10] https://en.wikipedia.org/wiki/Ischemia
[11] https://www.healthline.com/health/thallium-stress-test
[12] https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761
[13] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
[14] https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/
[15] https://medium.com/@moussadoumbia_90919/elbow-method-in-supervised-learning-optimal-k-value-99d425f229e7
[16] https://towardsdatascience.com/shallow-neural-networks-23594aa97a5
[17] https://scipython.com/blog/a-shallow-neural-network-for-simple-nonlinear-classification/
[18] Lecture Slide 8
[19] "Machine learning: A probabilistic perspective" by K. Murphy
[20] https://www.youtube.com/watch?v=_PwhiWxHK8o
[21] https://www.codecademy.com/articles/normalization

Appendix – Python Code of the Project

```
#!/usr/bin/env python

# coding: utf-8


# In[16]:




import numpy as np

import pandas as pd #for Importing Data

import matplotlib.pyplot as plt #for plotting

import seaborn as sns #for plotting

from ttictoc import tic,toc

from random import randrange

import random

from math import sqrt

from math import e


dt = pd.read_csv('heart.csv') #Edit accordingly

dt.columns = ['Age', 'Sex', 'Chest_pain_type', 'Resting_blood_pressure', 'Cholesterol', 'Fasting_blood_sugar', 'Rest_ecg', 'Max_heart_rate_achieved',

    'Exercise_induced_angina', 'St_depression', 'St_slope', 'Num_major_vessels', 'Thalassemia', 'Target']

dt.head()#For cheking



# In[3]:



dt.dtypes #As we can see some attributes must be of categorical value. However they are represented as integers.

#We will create dummy variables for categorical variables later.



# In[3]:
```

```
dt.describe() #Checking Properties of the Data


# In[4]:


dt.Target.value_counts() #it shows the number of patients who has heart disease and patients who doesn't have heart disease


# In[5]:


# plot histograms for each variable
dt.hist(figsize = (12, 12))
plt.show()


# In[6]:


#Produce Correlation Matrix to see how each attribution affects each other
plt.figure(figsize=(15,15))
sns.heatmap(dt.corr(),annot=True, linewidths=0.5, fmt='.2f', cmap="coolwarm")
plt.show()
#It can be seen that attributes are not linearly independent of each other.


# In[17]:


#Now we need to create dummy variables for categorical variables.
```

```
#We need to first decide on which attributes are the categorical variables

categorical_val = []

noncategorical_val = []

for column in dt.columns:

    if len(dt[column].unique()) <= 5:  #Educated Guess :D

        categorical_val.append(column)

    else:

        noncategorical_val.append(column)


print("Categorical Values:")

print(categorical_val)

print(f" Total Number : {len(categorical_val)}")

print('<==================================================================
========================>')

print("Non-Categorical Values:")

print(noncategorical_val)

print(f" Total Number : {len(noncategorical_val)}")



# In[8]:



#So, there are 9 categorical variables and 5 continuous variables. While continuous variables is already quantified,

#we have to quantify categorical variables as well. This is done by dummy coding.

#But firstly, lets see how continuous variables are related to target.

import plotly.express as px #For 3d data plot

fig = px.scatter_3d(dt, x='Cholesterol', y='Max_heart_rate_achieved', z='Age', size='St_depression',

              color='Target',opacity=1)

fig.show()

#Couldn't fit in Resting Blood Pressure



# In[9]:
```

sns.pairplot(dt,hue='Target',diag_kind="hist")

#With this plot, we can see the correlations better!

# In[18]:

categorical_val.remove('Target')

print(categorical_val)

# In[19]:

dataset = pd.get_dummies(dt, columns = categorical_val) #Using Panda's get_dummies command to convert categorical values to dummy variables.

dataset.describe()

# In[20]:

dataset.rename(columns={'Sex_0':'Female'}, inplace=True)

dataset.rename(columns={'Sex_1':'Male'}, inplace=True)

dataset.head()

# In[21]:

#Feature Scaling for better optimization

#Standardization

```
#We only need to standardize continuous features!


mean = dataset[['Age','Resting_blood_pressure','Cholesterol','Max_heart_rate_achieved','St_depression']].mean(axis=0)

dataset[['Age','Resting_blood_pressure','Cholesterol','Max_heart_rate_achieved','St_depression']] -= mean

std = dataset[['Age','Resting_blood_pressure','Cholesterol','Max_heart_rate_achieved','St_depression']].std(axis=0)

dataset[['Age','Resting_blood_pressure','Cholesterol','Max_heart_rate_achieved','St_depression']] /= std


dataset.head()




# In[7]:


# Now we will define other functions


def cross_validation(dataset, n_folds):  #Divide the dataset
    dcopy=dataset.copy()
    y = dcopy.Target
    X = dcopy.drop('Target', axis=1)
    dataset_splitX = []
    dataset_splitY = []
    X_copy = X.values.tolist()
    Y_copy = y.values.tolist()
    fold_size = int(len(dataset) / n_folds)
    for i in range(n_folds):
        foldX = list()
        foldY = list()
        while len(foldX) < fold_size:
            index = randrange(len(X_copy))
            foldX.append(X_copy.pop(index))
            foldY.append(Y_copy.pop(index))
        dataset_splitX.append(foldX)
        dataset_splitY.append(foldY)
    return dataset_splitX,dataset_splitY


def Accuracy(Y_real,Y_predicted): #Accuracy test
    correct = 0
    for i in range(len(Y_real)):
        if Y_real[i] == Y_predicted[i]:
            correct += 1
    return correct / float(len(Y_real)) * 100.0



#For kNN


def Distance_Norm(Vector1, Vector2): #Creating a function to find the norm of difference between two vectors.
    V1=np.array(Vector1)
    V2=np.array(Vector2)
    V=sum((V1-V2)**2)
    Distance=sqrt(V)
    return Distance



def get_Neigh(X_train,Y_train,X_single,num_Neigbour): #Function for getting the closest vectors for a single point and making a list like that
    distances=[]
    for k in range(len(X_train)):
        getDistance=Distance_Norm(X_train[k], X_single)
        distances.append((X_train[k],Y_train[k],getDistance))
    distances=sorted(distances,key=lambda x: x[2])#Sorting the List according to the shortest distance
    neighboursX=[]
    neighboursY=[]
    for i in range(num_Neigbour):
        neighboursX.append(distances[i][0])
        neighboursY.append(distances[i][1])
    return neighboursX,neighboursY


def predict_Y(X_train,Y_train, X_single, num_Neigbour):
    neighborsX,neighboursY = get_Neigh(X_train, Y_train, X_single, num_Neigbour)
    counts=[]
    for i in neighboursY:
        counts.append(i)
    ones=counts.count(1)
    zeros=counts.count(0)
    if ones >= zeros:
        return 1
    else:
        return 0


def kNN(X_train,Y_train,X_test,num_Neigbour):  #kNN algorithm with the help of other defined functions
    Y_predictions=[]
    for row in X_test:
        predictedY=predict_Y(X_train,Y_train, row, num_Neigbour)
        Y_predictions.append(predictedY)
    return Y_predictions



#Test for kNN algorithm
#In KNN, finding the value of k is not easy.
#A small value of k means that noise will have a higher influence on the result and a large value make it computationally expensive.
#k=sqrt(N) where N is the number of feature vectors


"""
dataset_copy=dataset.copy()
tic()
Scores = algorithm_evaluation(dataset_copy,6,kNN,7)
```

```
elapsed = toc()
print('Elapsed time:',elapsed)
print('<=================================================================>')
print('Scores: %s' % Scores)
print('Mean Accuracy: %.3f%%' % (sum(Scores)/float(len(Scores))))

#Elbow method for choosing k
Results=[]
for k in range(30):
    dataset_copy=dataset.copy()
    Scores = algorithm_evaluation(dataset_copy,6,kNN,(k+1))
    Result=(sum(Scores)/float(len(Scores)))
    Results.append(Result)

plt.plot(Results)
print(Results.index(max(Results)))
"""

# In[8]:



# Support Vector Machine



def gradient_function(W, X, Y, C):
    summ = 0
    for i in range(0, len(X)):
        if (1 - (Y[i] * np.dot(W, X[i]))) > 0:
            summ += (C * Y[i] * X[i])

    #  result of gradient of the objective function
    return W - summ
```

```
# stochastic gradient descent
def gradient_descent(maxR, gamma, X, Y, C, tol):
    # give weights
    W = np.zeros(len(X[0]))
    for k in range(0, maxR):
        # shuffle the list
        shuffle_list = list(zip(X,Y))
        random.shuffle(shuffle_list)
        X, Y = zip(*shuffle_list)

        grad = gradient_function(W, X, Y, C)
        diff = -grad * gamma

        if np.all(np.abs(diff) <= tol):
            return W

        else:
            W = W + diff
    return W

def svm_predict(X,W):
    results = []
    for col in X:
        ans = np.dot(W,col)
        if ans >=0:
            results.append(1)
        else:
            results.append(-1)
    return results



# Number of max repetitions for gradient descent
maxR = 200
# Learning rate of the gradient descent
```

```
gamma = 0.0001
# Tolerance to check if the gradient converges
tol = 1e-06



def SVM(X_train, Y_train, X_test, C, maxR, gamma, tol):
    for yi in range(len(Y_train)):
        if Y_train[yi] == 0:
            Y_train[yi] = -1
    X_test = np.array(X_test)
    X_train = np.array(X_train)
    Y_train = np.array(Y_train)
    # Add 1 to every row for bias term
    X_train = np.insert(X_train,0,1,axis = 1)
    X_test = np.insert(X_test,0,1,axis = 1)
    # get the weights of SVM
    W = gradient_descent(maxR, gamma, X_train, Y_train, C, tol)
    # for train accuracy
    train_res = svm_predict(X_train, W)
    # predict the test values
    result = svm_predict(X_test, W)
    return train_res, result



# In[55]:



#For Shallow Neural Network


#Initialize Network
def initialization_of_network(n_inputs, n_hidden, n_outputs):
    total_network=[]
    hidden_layers=[{"Weights": [random.random() for i in range(n_inputs + 1)]} for i in range(n_hidden)] #Extra Weight for bias
    output_layers = [{'Weights':[random.random() for i in range(n_hidden + 1)]} for i in range(n_outputs)] #Extra Weight for bias
    #So, first component is the weights of the hidden layer
    total_network.append(hidden_layers)
    #So, second component is the weights of the output layer
    total_network.append(output_layers)
    return total_network



#Forward Propagation
def neuron_activation(inputs,neuron_weights):
    activation=neuron_weights[-1]
    for i in range(len(inputs)):
        activation += inputs[i]*neuron_weights[i]
        transfer=1/(1 + e**(-activation))
    return transfer  #This is for each neuron

#Activation function is the sigmoid function!
#output = 1 / (1 + e^(-activation))

'''
def transfer_func(activation_output):
    transfer=1/(1 + e**(-activation_output))
    return transfer
'''
def forward_propagation(network, inputs):
    inputs=inputs
    for layer in network:
        new_input=[]
        for neuron in layer:
            weights=neuron["Weights"]
            '''
            activation=neuron_activation(inputs,weights)
            neuron["Outputs"]=transfer_func(activation)
            '''
            neuron["Outputs"]=neuron_activation(inputs,weights)
            new_input.append(neuron["Outputs"])
            inputs=new_input
    return inputs #Gives the final output


#Back Propagate Error
def transfer_derivative(output):
    return output * (1.0 - output) #Derivative of sigmoid


def backward_propagation_error(network, expected_output):
    #We will go backwards
    for index in reversed(range(len(network))):
        layer=network[index]
        errors=[]
        if index == len(network)-1:
            for neuron in layer:
                errors.append(expected_output-neuron["Outputs"])
        else:
            for j in range(len(layer)):
                error=0.0
                for neuron in network[index+1]:
                    error += (neuron["Weights"][j]*neuron["Delta"])
                errors.append(error)
        for k in range(len(layer)):
            neuron=layer[k]
            neuron['Delta'] = errors[k] * transfer_derivative(neuron['Outputs'])


#Train Network
#We have to update the weights, we do that by weight = weight + learning_rate (new variable) * error(stored in delta) * input
def update_weights(network, input_row , l_rate):
    for index in range(len(network)):
        inputs=input_row
        if not index == 0:
            inputs = [neuron['Outputs'] for neuron in network[index - 1]]
        for neuron in network[index]:
            for k in range(len(inputs)):
                neuron["Weights"][k]=(1-
0.003*l_rate)*neuron["Weights"][k]+(l_rate*neuron["Delta"]*inputs[k])
            neuron['Weights'][-1] = (1-0.003*l_rate)*neuron['Weights'][-1]+l_rate * neuron['Delta'] #For
bias


#
def Network_Train(network,train_input,l_rate,n_epoch,train_output):
    for epoch in range(n_epoch):
        total_error=0
        for index in range(len(train_input)):
            input_row=train_input[index]
            expected_output=train_output[index]
            output=forward_propagation(network, input_row)
            total_error += (expected_output-output[0])**2
            backward_propagation_error(network, expected_output)
            update_weights(network, input_row, l_rate)
        # error=sqrt(total_error)
        # print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, error))



def Prediction(network,input_row):
    output=forward_propagation(network, input_row)
    if output[0] >0.5:
        return 1.0
    else:
        return 0.0


#Stochastic Gradient Descent
```

```
def back_propagation(trainX, trainY, testX, l_rate, n_epoch, n_hidden):
    n_inputs = len(trainX[0])
    n_outputs = 1
    network = initialization_of_network(n_inputs, n_hidden, n_outputs)
    Network_Train(network, trainX, l_rate, n_epoch, trainY)
    train_predictions = []
    for row in trainX:
        prediction = Prediction(network, row)
        train_predictions.append(prediction)
    test_predictions = []
    for row in testX:
        prediction = Prediction(network, row)
        test_predictions.append(prediction)
    return train_predictions, test_predictions




# In[ ]:


# split test set from dataset
dd = dataset.copy()
train_df = dd.sample(frac = 0.7)
test_df = dd.drop(train_df.index)
x_val, y_val = cross_validation(train_df.copy(), 6)

train_acc_list1 = []
test_acc_list1 = []
N = np.arange(8,25,2)
for n in N:
    train_acc1 = 0
    test_acc1 = 0
    for ind in range(0, len(x_val)):
        X_test = x_val[ind]
        Y_test = y_val[ind]
        X_train = []
        Y_train = []
        for i in range(len(x_val)):
            if i != ind:
                X_train += x_val[i]
                Y_train += y_val[i]
        trainp, testp = back_propagation(X_train, Y_train, X_test, 0.7, 120, n)
        train_acc1 += Accuracy(Y_train, trainp)
        test_acc1 += Accuracy(Y_test, testp)
    trainacc = train_acc1 / len(x_val)
    testacc = test_acc1 / len(x_val)
    train_acc_list1.append(trainacc)
    test_acc_list1.append(testacc)
plt.plot(N, train_acc_list1)
plt.plot(N, test_acc_list1)
maxtest1 = max(test_acc_list1)
optimal_n = test_acc_list1.index(maxtest1)
print(f"The optimal value of n_hidden is: {N[optimal_n]}")

plt.figure
plt.plot(N, train_acc_list1)
plt.plot(N, test_acc_list1)
plt.xlabel("n_hidden")
plt.ylabel("%acc")
plt.title("Parameter tuning of n_hidden for SNN")
plt.legend(['Train accuracy', 'Test accuracy'])


# find the optimal C, the lagrange multiplier of SVM
# cross validation to find the best value of C between 5 and 150
for fold in y_val:
    for yi in range(len(fold)):
        if fold[yi] == 0:
            fold[yi] = -1
```

```
train_acc_list2 = []
test_acc_list2 = []
R = np.arange(5, 155, 5)
for c in R:
    train_acc2 = 0
    test_acc2 = 0
    for ind in range(0, len(x_val)):
        X_test = x_val[ind]
        Y_test = y_val[ind]
        X_train = []
        Y_train = []
        for i in range(len(x_val)):
            if i != ind:
                X_train += x_val[i]
                Y_train += y_val[i]
        X_test = np.array(X_test)
        Y_test = np.array(Y_test)
        X_train = np.array(X_train)
        Y_train = np.array(Y_train)
        # Add 1 to every row for bias term
        X_train = np.insert(X_train,0,1,axis = 1)
        X_test = np.insert(X_test,0,1,axis = 1)

        W = gradient_descent(maxR, gamma, X_train, Y_train, c, tol)
        trainres = svm_predict(X_train, W)
        train_acc2 += Accuracy(Y_train, trainres)
        testres = svm_predict(X_test, W)
        test_acc2 += Accuracy(Y_test, testres)
    trainacc = train_acc2 / len(x_val)
    testacc = test_acc2 / len(x_val)
    train_acc_list2.append(trainacc)
    test_acc_list2.append(testacc)

plt.figure
plt.plot(R, train_acc_list2)
plt.plot(R, test_acc_list2)
plt.xlabel("C")
plt.ylabel("%acc")
plt.title("Parameter tuning of C for SVM")
plt.legend(['Train accuracy', 'Test accuracy'])

maxtest2 = max(test_acc_list2)
optimal_c = test_acc_list2.index(maxtest2)
print(f"The optimal value of C is: {R[optimal_c]}")


# In[ ]:


# one test to rule them all

def final_results(train_df,test_df):

    train_df1 = train_df.copy()
    train_setY1 = train_df1.Target.values.tolist()
    train_setX1 = train_df1.drop('Target', axis=1).values.tolist()
    train_df2 = train_df.copy()
    train_setY2 = np.array(train_df2.Target.values.tolist())
    train_setX2 = np.array(train_df2.drop('Target', axis=1).values.tolist())
    train_df3 = train_df.copy()
    train_setY3 = train_df3.Target.values.tolist()
    train_setX3 = train_df3.drop('Target', axis=1).values.tolist()

    test_df1 = test_df.copy()
    test_setY1 = test_df1.Target.tolist()
    test_setX1 = test_df1.drop('Target', axis=1).values.tolist()
    test_df2 = test_df.copy()
    test_setY2 = np.array(test_df2.Target.values.tolist())
    test_setX2 = np.array(test_df2.drop('Target', axis=1).values.tolist())
```

```
test_df3 = test_df.copy()
test_setY3 = test_df3.Target.values.tolist()
test_setX3 = test_df3.drop('Target', axis=1).values.tolist()

# test of kNN
tic()
kNN_predict = kNN(train_setX1, train_setY1, test_setX1, 7)
kNN_time = toc()
kNN_accuracy = Accuracy(test_setY1, kNN_predict)
print("kNN elapsed time:", kNN_time)
print("kNN accuracy:", kNN_accuracy)

# test of SVM
tic()
svm_train_predict, svm_test_predict = SVM(train_setX2, train_setY2, test_setX2, R[optimal_c], maxR, gamma, tol)
svm_time = toc()
for yi in range(len(svm_test_predict)):
    if svm_test_predict[yi] == -1:
        svm_test_predict[yi] = 0
svm_train_accuracy = Accuracy(train_setY2, svm_train_predict)
svm_test_accuracy = Accuracy(test_setY2, svm_test_predict)
print("SVM elapsed time:", svm_time)
print("SVM train accuracy:", svm_train_accuracy)
print("SVM test accuracy:", svm_test_accuracy)

# test of SNN
tic()
snn_train_predict, snn_test_predict = back_propagation(train_setX3, train_setY3, test_setX3, 0.7, 200, N[optimal_n]) #Used heuristics for these numbers
snn_time = toc()
for yi in range(len(train_setY3)):
    if train_setY3[yi] == -1:
        train_setY3[yi] = 0
    snn_train_accuracy = Accuracy(train_setY3, snn_train_predict)
    snn_test_accuracy = Accuracy(test_setY3, snn_test_predict)
    print("SNN elapsed time:", snn_time)
    print("SNN train accuracy:", snn_train_accuracy)
    print("SNN test accuracy:", snn_test_accuracy)

final_results(train_df,test_df)
```