Ahmet Faruk Çetinkaya 22103752
Berkay Yalman 22102859
Group 36

EEE-485 Spring 2024 Project Final Report
Machine Learning for Discriminating Primary Gamma Signals from Atmospheric Hadronic Shower Background in Cosmic Ray Imaging

Introduction and Problem Description

Primary gamma rays are subject to many astrophysical studies. But observing signals of primary gamma rays is a challenging process due to background noises. Cosmic rays' constant interaction with earth's atmosphere creates a background of secondary particles. These particles can imitate the signals from primary gamma rays. Therefore, distinguishing primary gamma signals and hadronic shower background is an important task in cosmic ray imaging. In this project, our goal is to develop a binary classification system that distinguishes real gamma signals from hadrons (background noise). 3 Machine learning algorithms will be used to accomplish this task. They are K- Nearest Neighbors (k-NN), Neural Networks, Support Vector Machines (SVM). These methods will be implemented in Python. The success of these methods will be monitored.

Dataset Description

The dataset was obtained from the UCI Machine Learning Repository (https://archive.ics.uci.edu/dataset/159/magic+gamma+telescope). It has 19020 instances (12332 gammas, 6688 hadrons) and every instance has 10 features. The classes are hadron(h) and gamma (g). The features are: fLength: major axis of ellipse [mm] ,fWidth: minor axis of ellipse [mm] , fSize: 10-log of sum of content of all pixels [in #phot], fConc: ratio of sum of two highest pixels over fSize [ratio] ,fConc1: ratio of highest pixel over fSize [ratio] , fAsym: distance from highest pixel to center, projected onto major axis [mm] , fM3Long: 3rd root of third moment along major axis [mm] , fM3Trans: 3rd root of third moment along minor axis [mm], fAlpha: angle of major axis with vector to origin [deg], fDist: distance from origin to center of ellipse [mm] , class: g,h - gamma (signal), hadron (background)

Simulation Setup

The python language was used for training and observing performance of results. Pandas library was used to capture data from csv file. NumPy is used for many matrix and calculation operations. Matplotlib and Seaborn were also used for visualization. Furthermore, several feature engineering and preprocessing tasks have been implemented to enhance the quality of the data.

1. Dataset Preprocessing

Preprocessing is a crucial step that should be applied to data to enhance the accuracy of interpretations and predictions.

a) Standardization: In this dataset, it is apparent that the features are not standardized; they do not share a uniform format, which complicates the interaction and processing between features and outcomes. This can be problematic for methods we have like Neural Networks and Support Vector Machines since they are needed for feature scaling. Therefore, we normalized the data. The normalization process is carried out using the following calculation:

$$\mu = \frac{1}{n}\sum_{i=1}^{n}(x_i)\,,\ \sigma = \sqrt{\frac{1}{n}\cdot\sum_{i=1}^{n}(x_i-\mu)} \qquad \text{Standardized Data}=\frac{x_i-\mu}{\sigma}$$

b) Addressing Data Imbalance: In our original dataset we have 12332 gamma, 6688 hadron labels. To eliminate this inequality, the first 5643 gamma instances were deleted from the data.

c) Feature engineering for Support Vector Machine: In the Support Vector Machine algorithm, we use a linear algorithm, but since our data set is probably non-linear, the accuracy is not very high. To increase efficiency, we tried to make the data behave as linear by doing some feature engineering. It is done by adding some multiplication of features, adding squared features to original feature vectors. What was done exactly is explained in the SVM performance evolution section.

2. Dataset Analysis

a) PCA: We run the PCA to address behavior of the data in terms of variation of features and correlation between features. We analyzed the total variation explained by principal components. PCA is carried out using the following calculation:

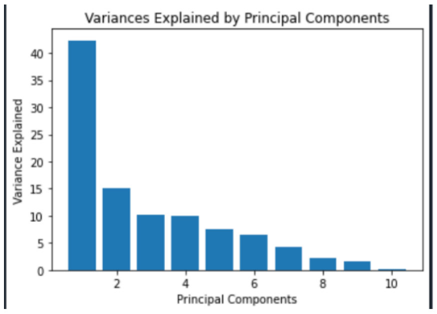
$$\Sigma = \frac{1}{n} X^T X$$

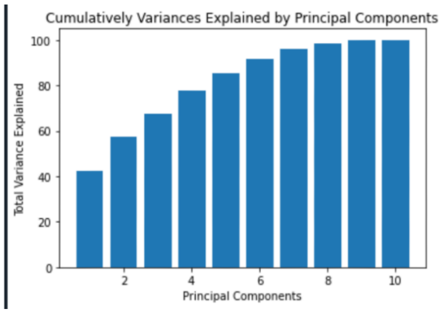
$$\Sigma u = \lambda u \implies (\Sigma - I\lambda)u = 0 \implies \det(\Sigma - I\lambda) = 0$$

$$\lambda_1 > \lambda_2 \cdots > \lambda_{10} \; are\ eigen\ values\ of\ covariance\ matrix$$

$$u_1, u_2, \cdots, u_{10} \; are\ eigenvectors\ correspoding\ to\ eigen\ values$$

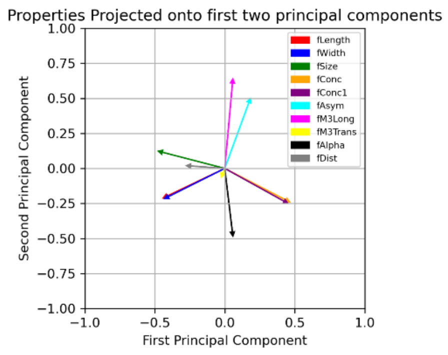
$$PVE(k) = \frac{\lambda_k}{\sum_{i=1}^{10} \lambda_i}, \; PVE(first\ k) = \sum_{j=1}^{k} PVE(j)$$



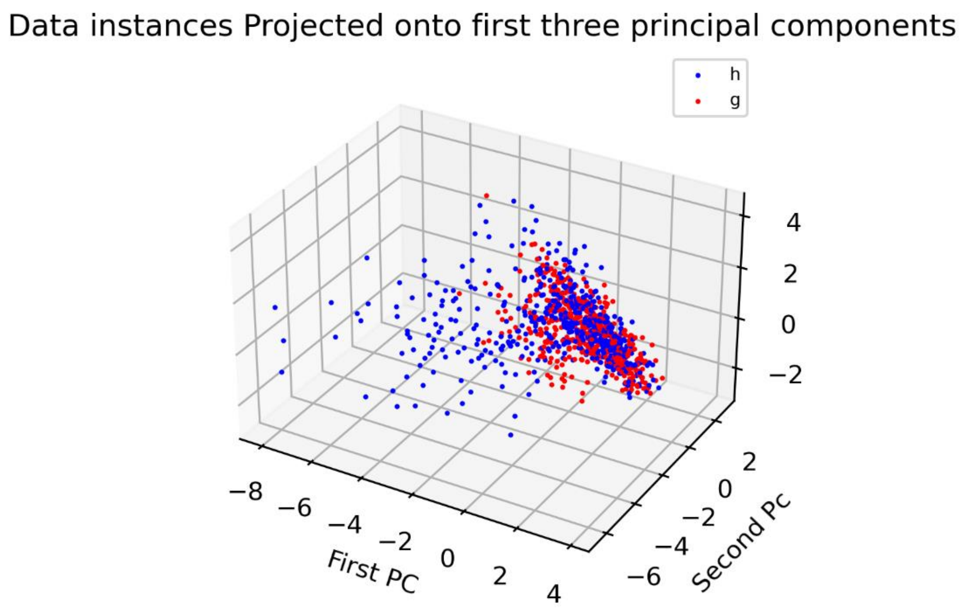


The first and second PC explains more than half of the variation and when last 2 PC excluded, rest can explain more than 90 percent of variation. The dimension of the data can be reduced to 8 if needed in some applications.

We projected features onto first two PC using $z_i = [u_1, u_2]^T \cdot x_i$:



First two features namely ’fLenght’ and ’fWidth’ have almost identical projections on first two PC. Same situations also apply to ’fConc’ and ’fConc1’. These features probably have a physical relationship in the real world.

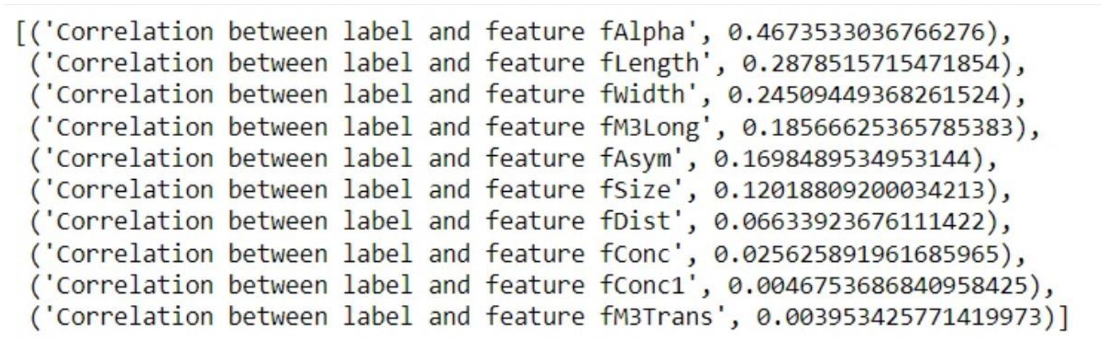


b) Feature Ranking via Correlation

We ranked the features according to their correlation among themselves and with label. Correlation between features is measured using covariance matrix which is calculated in the PCA analysis.

b-1 Correlation between features and labels

Correlation coefficients calculated using $R_j = \frac{Cov(x_j, y)}{\sqrt{Var(x_j) \cdot Var(y)}} = \frac{\sum_{i=0}^{n}(x_{ij}-\overline{x_j})(y_i-\overline{y})}{\sqrt{\sum_{i=0}^{n}(x_{ij}-\overline{x_j})^2 \cdot \sum_{i=0}^{n}(y_i-\overline{y})^2}}$

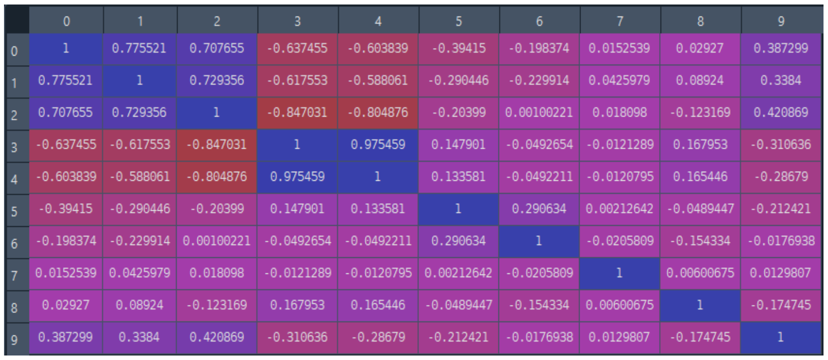


```
[('Correlation between label and feature fAlpha', 0.4673533036766276),
 ('Correlation between label and feature fLength', 0.2878515715471854),
 ('Correlation between label and feature fWidth', 0.24509449368261524),
 ('Correlation between label and feature fM3Long', 0.18566625365785383),
 ('Correlation between label and feature fAsym', 0.1698489534953144),
 ('Correlation between label and feature fSize', 0.12018809200034213),
 ('Correlation between label and feature fDist', 0.06633923676111422),
 ('Correlation between label and feature fConc', 0.025625891961685965),
 ('Correlation between label and feature fConc1', 0.0046753686840958425),
 ('Correlation between label and feature fM3Trans', 0.003953425771419973)]
```

‘fAlpha’ and ’fLength’ have the highest correlation with the label and ’fM3Trans’ and ‘fConc1’ have the lowest correlation with the label.

b-2 Correlation between features

Features ’fConc’ and ’fConc1’ showed the highest correlation with 0.975 which was also predicted in the PCA section. Also, the features ’fLenght’,’fWidth’ and ’fSize’ showed correlation among themselves



	0	1	2	3	4	5	6	7	8	9
0	1	0.775521	0.707655	-0.637455	-0.603839	-0.39415	-0.198374	0.0152539	0.02927	0.387299
1	0.775521	1	0.729356	-0.617553	-0.588061	-0.290446	-0.229914	0.0425979	0.08924	0.3384
2	0.707655	0.729356	1	-0.847031	-0.804876	-0.20399	0.00100221	0.018098	-0.123169	0.420869
3	-0.637455	-0.617553	-0.847031	1	0.975459	0.147901	-0.0492654	-0.0121289	0.167953	-0.310636
4	-0.603839	-0.588061	-0.804876	0.975459	1	0.133581	-0.0492211	-0.0120795	0.165446	-0.28679
5	-0.39415	-0.290446	-0.20399	0.147901	0.133581	1	0.290634	0.00212642	-0.0489447	-0.212421
6	-0.198374	-0.229914	0.00100221	-0.0492654	-0.0492211	0.290634	1	-0.0205809	-0.154334	-0.0176938
7	0.0152539	0.0425979	0.018098	-0.0121289	-0.0120795	0.00212642	-0.0205809	1	0.00600675	0.0129807
8	0.02927	0.08924	-0.123169	0.167953	0.165446	-0.0489447	-0.154334	0.00600675	1	-0.174745
9	0.387299	0.3384	0.420869	-0.310636	-0.28679	-0.212421	-0.0176938	0.0129807	-0.174745	1

about 0.7. Correlation of first two features was predicted in the PCA section but the correlation of third feature with first two was not visible in the PCA.

Implemented Algorithms

1) Neural Networks (NN)

Neural networks can capture complex nonlinear relationships between features and labels [1]. They also have the advantage of parallel processing in both training and use. These are the reasons why we choose this method. This method utilizes perceptron's that are grouped in layers, and layers are cascaded from input to output. Perceptron is a composite function that has linear function inside and nonlinear function outside [2]. Cascading perceptron's enables neural network to be effective on nonlinear relations. Layer numbers and perceptron count in a layer are hyperparameters which can be optimized to get the best results. Evaluation of and plots of hyperparameters will be given in the k-fold section.

Perceptron:

$$v = \sum_{i=0}^{d^l} w_i x_i \ , \ output = \phi(v)$$

Φ is the activation function of the perceptron, generally a nonlinear function [3]. We used the RELU and logistic functions in our neural network. RELU is used in hidden layers and logistic (Φ) used in output layer.

$$RELU(x) = x \, if \, x \geq 0 \, otherwise \, 0, \ \Phi(x) = \frac{1}{1+e^{-x}}$$

Layers:

We used a two hidden layer neural network; therefore, we have three set of weights namely input-layer1 layer1-layer2 layer2-output. Output of neural network calculated as follows:

$$y = \Phi\left(W_{2-o} \cdot \left(RELU \cdot \left(W_{1-2} \cdot RELU(W_{i-1} \cdot x_i)\right)\right)\right)$$

Then prediction is decided comparing y value to a threshold value:

$$y_{pred} = 1 \, if \, y \geq threshold, \, otherwise \, 0$$

Training: To train the neural network we used backpropagation method and mini batch gradient descent method. The sizes of mini batches are determined by k-fold, one batch contains n/k data instance and learning rate is multiplied by a 1/k factor. Loss function is chosen as sum of squares. Backpropagation is calculated as follows:

J is error function; v is induced local field which is the linear part of perceptron [3]

$$\delta_j^l = -\frac{\partial J(y, y_{pred})}{\partial v_j^l}, \ \frac{\partial J(y, y_{pred})}{\partial w_{ij}^l} = \frac{\partial J(y, y_{pred})}{\partial v_j^l}\frac{\partial v_j^l}{\partial w_{ij}^l} = \delta_j^l \cdot x_i^{l-1}$$

$$\delta_i^{l-1} = -\frac{\partial J(y, y_{pred})}{\partial v_i^{l-1}} = \sum_{j=1}^{d(l)} \delta_j^l \cdot w_{ij}^l \cdot \frac{\partial x_i^{l-1}}{\partial v_i^{l-1}}$$

$$w_{ij}^l = w_{ij}^l - \frac{\eta}{k} \frac{\partial J(y, y_{pred})}{\partial w_{ij}^l}$$

2) K- Nearest Neighbors (kNN)

KNN (k-nearest neighbors) is a simple and crucial algorithm which can be used for classification problems in machine learning [4]. That's why we choose this method. This is a supervised machine learning algorithm because it is based on labeled data to train a function that can generate a suitable output [4]. We can explain this algorithm in a simple way as follows. We are choosing a test data instance and calculating squared distances between this data and all other training data. This requires vector operation since we have multidimensional data. Subsequently, we are comparing these distances and selecting k smallest ones. Using indexes of this k vector, corresponding outputs are chosen from training data. Finally, the mean of these outputs gives us the probability of being class 1 ,i.e, gamma for test data instance. Normally, If the number of the closest data is more, the algorithm can also run by matching the label of the new data with it, but we preferred to use a probabilistic approach. After finding the probability, the label of the new data instance is determined according to the threshold chosen. The process can be illustrated in the following figures.

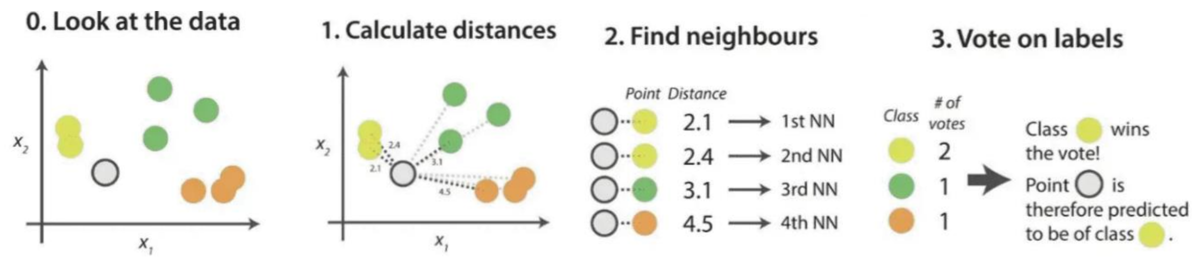


Figure 1: The illustration of K-NN Algorithm [5]

Feature scaling is essential for this method because it relies on Euclidean distance to predict classes. If the feature vector isn't scaled, the distance calculation could be disproportionately influenced by features whose range is broader. Therefore, normalization is crucial to ensure balanced contributions from all features. A major drawback of the algorithm is its computational intensity, as it involves calculating the Euclidean distance between each vector in the dataset and the target vector. As the dataset size grows, the algorithm's speed diminishes, negatively impacting its efficiency. Moreover, another significant issue with the kNN algorithm is the absence of a definitive method for selecting the optimal *k* value. This issue also arose in our scenario, which we will explore further when discussing the accuracy of algorithms. Due to the lack of a clear-cut approach for determining *k*, we must rely on techniques like the elbow method to address this problem by running k-fold cross validation [1].

3) Support Vector Machines (SVM)

SVM is a type of supervised learning algorithm used in machine learning that classifies data by identifying the best possible boundary that separates different classes [6]. This optimal boundary is determined by maximizing the margin between the closest points of the classes, which are known as support vectors. In other words, while the perceptron algorithm ensures the identification of a separating hyperplane when one exists, SVMs take this a step further by identifying the hyperplane that not only separates the classes but also maximizes the margin between them [7]. That's why we chose this algorithm. Visualization of SVM can be seen in Figure 2.

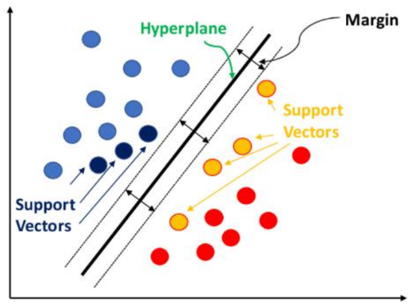


Figure 2: The illustration of Support Vector Machine [8]

In general, there are two different SVMs which are SVM with kernel and linear SVM. A linear SVM is utilized in this project. The decision to use a linear SVM was made to lower both the computational demands and the time required to train the model, providing a simpler and faster solution while forgoing the ability to separate data in non-linear ways. A l2 norm regularization was added to reduce complexity. Algorithm can be simply explained by mathematical models in the following. We need to give –1 and 1 to labels (+1 for gamma, -1 for hadron). When $\underline{w}.\underline{x_i} - b \geq 1$ y is 1, while $\underline{w}.\underline{x_i} - b \leq -1$ y is –1. This can be combined as $y_i\left(\underline{w}.\underline{x_i} - b\right) \geq 1$. For SVM, we have hinge loss function with regularization parameter which is in the following:

$$L = 0 \qquad if\ yi\left(\underline{w}.\underline{x}i - b\right) \geq 1$$

$$1 - y_i(w.x_i - b)\ otherwise$$

With regularization $L = \lambda < \left|\underline{w}\right|^2 + \frac{1}{n} \sum_{i=1}^{n} \max\left(0{,}1 - y_i\left(\underline{w}.x_i - b\right)\right)$

The loss is minimized with respect to the weight vector to find best weights, similarly to the NN algorithm. This minimization is achieved through Stochastic Gradient Descent (SGD), where the gradient of the loss function is computed, and the weights are adjusted based on this gradient. This process continues iteratively until the number of iterations ends. ($\underline{w}_{n+1} = \underline{w}_n + \gamma\Delta L$)

The computation time for the weight vector in SVM can vary based on the regularization parameter, learning rate, and the number of iterations performed during gradient descent. Finally, we can find labels for new data using learned weights. Since this is a linear SVM, we will try to make some feature engineering to try make our non-linear data more linear as mentioned earlier.

Outcomes of Simulations and Analysis of Performance

First of all, each method will be optimized in itself. For this, parameter tuning will be done using the k fold cross validation method. In the first report, we found the best parameters of each method. Secondly, we created train/test split the data. We trained each method with the same train set and looked at their performance with the same test set. In this way, we decided which method is the best.

Validation of Methods

k-Fold Cross Validation

Cross-validation provides insights into how well a model will perform on unseen data. It also helps to obtain a more precise evaluation of the model's predictive accuracy [9]. Therefore, we decided to use k-fold cross validation to select the best parameters for the methods. We used the whole data for cross validation and set the fold number as 6 or 10 or 100. The first step of k-NN algorithm is dividing dataset into k parts. The second step is using k-1 parts for training, 1 part for test purposing and finding accuracy for this setup. The third step is repeating the second step for k times by shifting. The last step is finding the average of each combination.

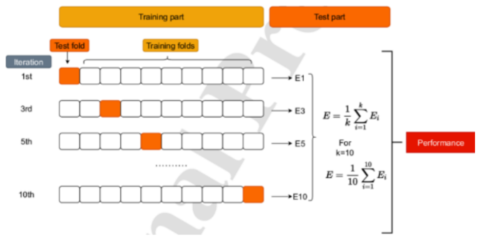
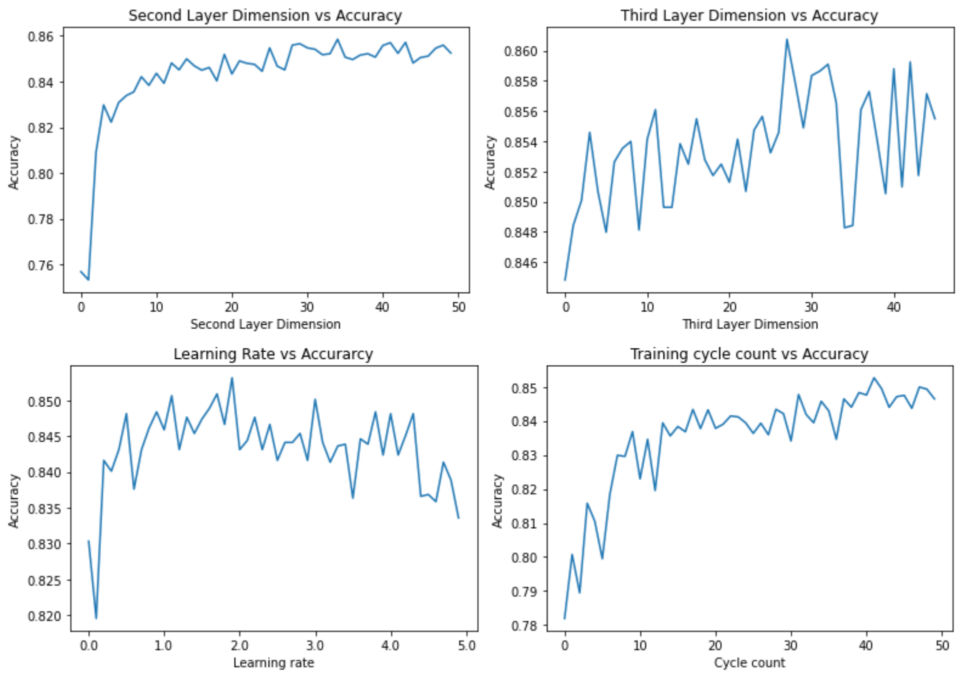


Figure 3: A summary of k-fold cross validation [10]

1) Neural Networks (NN)

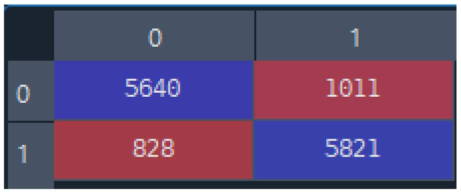
Using k-fold cross validations we measured the optimal values of hyperparameters second layer dimension, third layer dimension, learning rate and cycle count that is used to train weights. Cycle count is how many times algorithm runs over test data.



We found that both second- and third-layer dimensions accuracy gain converge around 30, further increasing these dimensions are unnecessary computation power.

Learning rate gives best accuracy at around 1 to 2. Lowering or increasing too much decreases accuracy.

The training cycle increases accuracy steadily until the 40 to 50 range, after that increasing just creates randomness. It is a costly hyperparameter to increase, it linearly increases training time when increased. Therefore, we chose the cycle count as 40 to keep required computation for training minimum.



	0	1
0	5640	1011
1	828	5821

Confusion matrix of the model after running over all data with k-cross validation. Confusion matrix shows that the model has no significant bias.

2) K- Nearest Neighbors (KNN)

The k-NN algorithm was executed, and accuracy was measured across various values of k (number of neighbors) by using k-fold cross validation. Firstly, probability threshold chosen as 0.6 and algorithm was run for 20 different k (number of neighbors). To observe the accuracy vs k relationship, a graph was created and is presented in Figure 4. It has been noted that the number of nearest neighbors chosen impacts the accuracy of the k-NN method. We can choose best k value from the graph. k value=10 had the highest accuracy. Hence, for the remainder of the project, we will use k=10. After finding ideal k value, we should find best threshold for separating being 1 and 0 probability. To observe the accuracy vs threshold, a graph was created and is presented in Figure 5. According to the graph best thresholds were 0.55, so we can use 0.55.

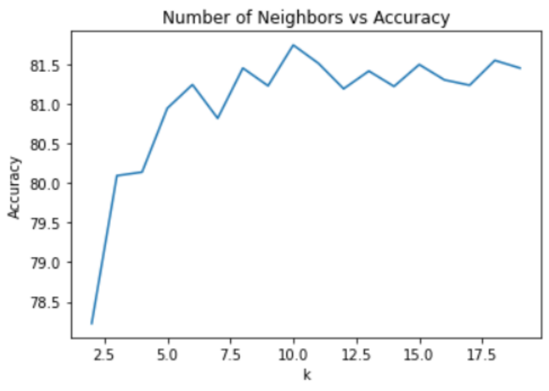


Figure 4: The plot of number of neighbors(k) vs accuracy

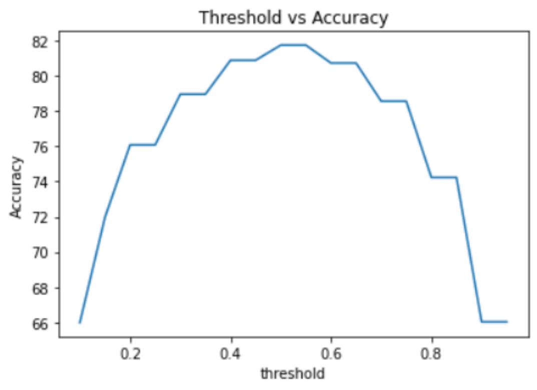


Figure 5: The plot of threshold vs accuracy

In conclusion, the best accuracy was found as 81.75 % with 10 nearest neighbor and 0.55 threshold. The execution time was 12 seconds, with the nearest number of neighbors set to 6 and the number of folds specified as 100. Even if we use vectorized operations when calculating distances in algorithm, it takes a little time since we use all 13376 data while running k-fold cross validation.

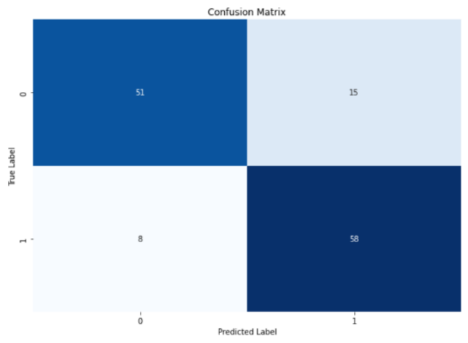


Figure 6: Confusion matrix of 10-nearest neighbor method

3) Support Vector Machines (SVM)

The SVM algorithm was executed, and accuracy was measured across various values of learning rate, regularization parameter and number of iterations by using k-fold cross validation. First of all, in order to select the ideal parameter values, the number of folds in k folds is taken as 3. The reason for this is that the SVM algorithm takes a lot of time. It passes all datas in for loop, we are not sure that this operation can be vectorized. However, we did not vectorize it since the model takes relatively short time to execute when compared execution times of models at train test split stage. After finding best values for learning rate, regularization parameter and number of iterations, we will increase k fold number to increase efficiency. Firstly, we will analyze relationship between regularization constant and accuracy by fixing learning rate and num of iterations. This relationship can be seen in figure 7. The best regularization parameter can be taken as 0.0001 since it is one of the values which have highest accuracy. Now, we can observe relationship between learning rate and accuracy by fixing other parameters. This can be seen in Figure 8.

The best learning rate can be taken as 0.0001 since it is best value in terms of accuracy. Finally, we can observe the effect of num of iterations by giving algorithm best parameters. This can be seen in Figure 9.

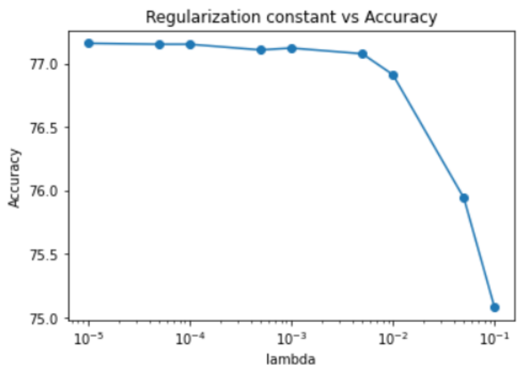


Figure 7: The plot of regularization constant vs accuracy

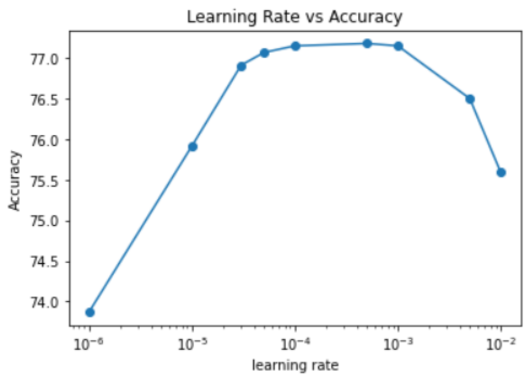


Figure 8: The plot of learning rate vs accuracy

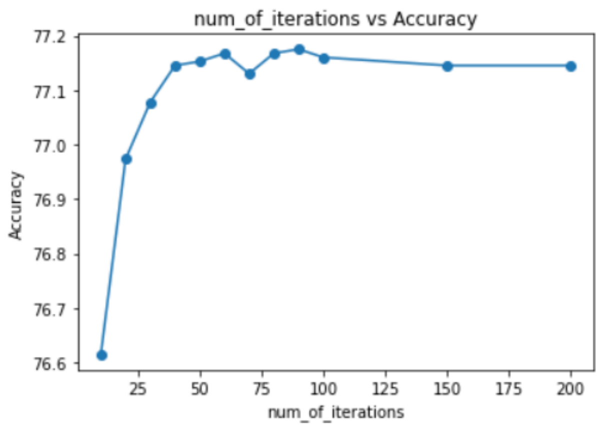


Figure 9: The plot of number of iterations vs accuracy

We can use elbow method to increase time efficiency by choosing iteration number is 50. Anything more will increase the accuracy slightly, but it will take more time. Now, we can increase k fold number to increase accuracy. When k fold=10 and with best parameters, accuracy was 77.1 %. As a mentioned before, we will do some feature engineering to try to cause the data to behave as linear.

Firstly, we add $(2th\,feature)^2$ to feature vector

$$\rightarrow X_i = x_{i0} + x_{i1} + x_{i2} + x_{i3} + x_{i4} + x_{i5} + x_{i6} + x_{i7} + x_{i8} + x_{i9} + (x_{i1})^2$$

$$\rightarrow\ Accuracy =\ 77.50$$

Secondly, we also add $(3th\,feature \cdot 4th\,feature)$ to feature vector

$$\rightarrow X_i = x_{i0} + x_{i1} + x_{i2} + x_{i3} + x_{i4} + x_{i5} + x_{i6} + x_{i7} + x_{i8} + x_{i9} + (x_{i1})^2 + (x_{i2} \cdot x_{i3})$$

$$\rightarrow\ Accuracy =\ 80.45$$

Thirdly, we also add $(5th\,feature \cdot 9th\,feature)$ to feature vector

$$\rightarrow X_i = x_{i0} + x_{i1} + x_{i2} + x_{i3} + x_{i4} + x_{i5} + x_{i6} + x_{i7} + x_{i8} + x_{i9} + (x_{i1})^2 + (x_{i4} \cdot x_{i8}) + (x_{i2} \cdot x_{i3})$$

$$\rightarrow\ Accuracy =\ 80.67$$

With a little feature engineering, we achieved an accuracy increase of 3 %. This value can be increased even more with different combinations. In conclusion, the best accuracy was found with some feature engineering as 80.67 % with learning rate=0.0001, regularization parameter=0.0001 and num iterations=50. The execution time was 50 seconds, with the number of folds specified as 10. Furthermore, a confusion matrix which shows distribution of predictions, created using the SVM method, is displayed below in Figure 10, providing a summary of the SVM algorithm's performance.

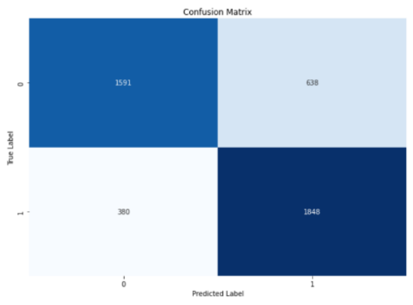


Figure 10: Confusion matrix for SVM algorithm

Comparison of Methods using Train/Test split data

We split the data randomly into the train and test sets as %80 and %20 respectively. We changed the standardizing method in preprocessing to prevent information leaking from train set to test set. We calculated the mean and std from the training data only. Subsequently we standardized both train and test data using these mean and std values. We also created feature engineered versions of the train and test sets to evaluate feature engineering performance of all methods. Results can be seen in Figure 11 below.

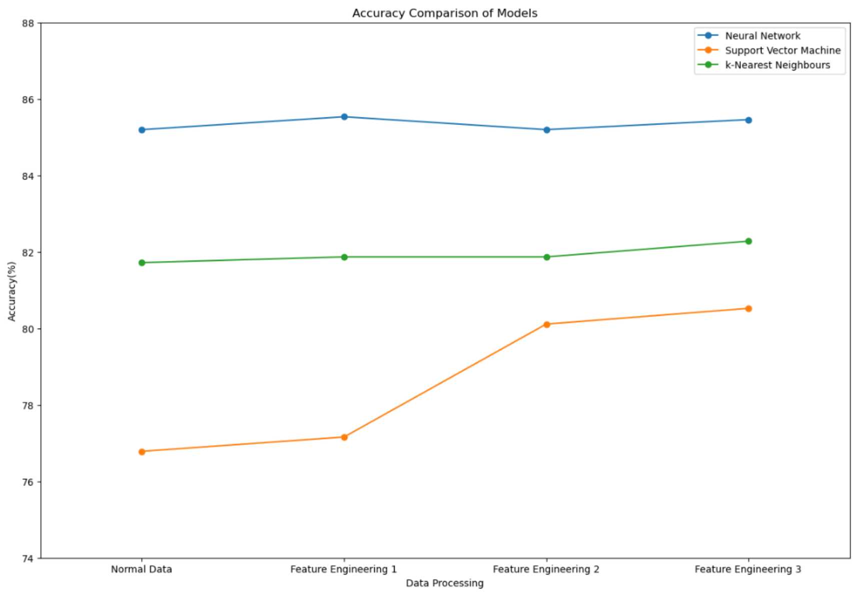


Figure 11: Accuracy comparison of methods

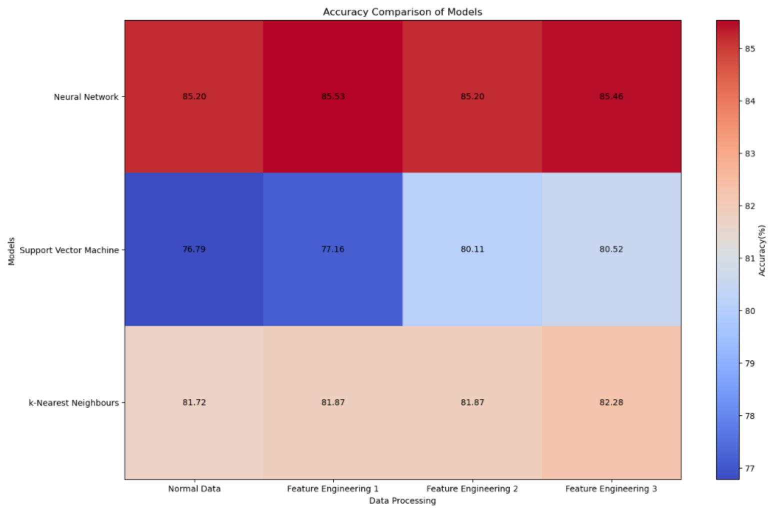


Figure 12: Accuracy comparison of methods

Neural Network and K-NN methods did not benefit from feature engineering as much as Support Vector Machine. The reason is that SVM algorithm is implemented by assuming data is linear however this is not the case. When we did feature engineering, we made data to behave nonlinearly, therefore SVM performance increased. On the other hand, Neural Network can learn nonlinear relations in the data inherently without external assistance. Therefore, feature engineering did not show notable improvements on the Neural Network. The k-NN does not inherently favor linear or nonlinear relation in the data set, furthermore, adding more dimensions to the data using feature engineering might deter the working of k-NN since it treats all dimensions equally. Overall Neural Network is the best performing algorithm with an accuracy of 85.

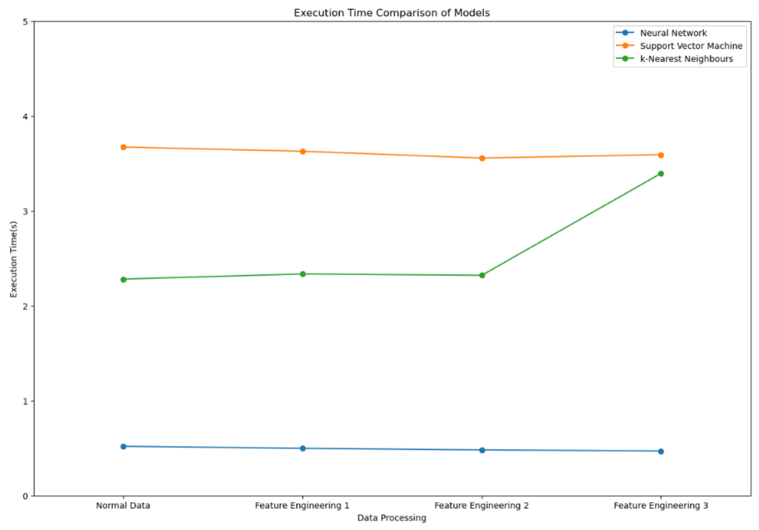
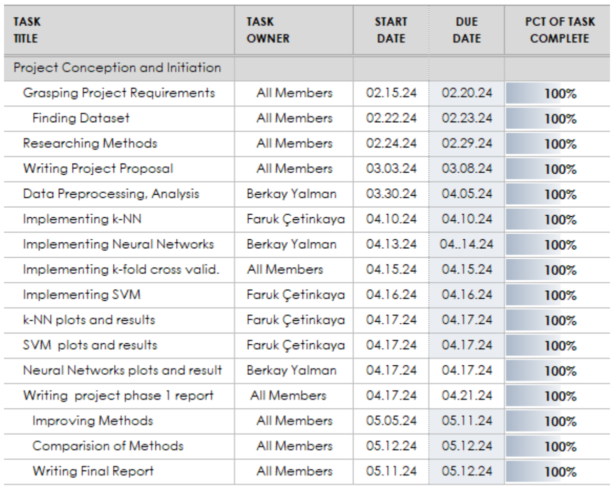


Figure 13: Execution time of models

Performance of the models in terms of execution times were observed. Again, the best performing model was Neural Network since it was highly vectorized and can calculate estimation of batches of the data at the same time using vectorization. Although k-NN is also vectorized it can’t calculate estimations in parallel unlike Neural Network, it calculates them one by one. Lastly Support Vector Machine has the worst performance since it was not fully vectorized.

Gantt Chart

Task Distribution between group members can be seen in Figure 10.



TASK TITLE	TASK OWNER	START DATE	DUE DATE	PCT OF TASK COMPLETE
Project Conception and Initiation				
Grasping Project Requirements	All Members	02.15.24	02.20.24	100%
Finding Dataset	All Members	02.22.24	02.23.24	100%
Researching Methods	All Members	02.24.24	02.29.24	100%
Writing Project Proposal	All Members	03.03.24	03.08.24	100%
Data Preprocessing, Analysis	Berkay Yalman	03.30.24	04.05.24	100%
Implementing k-NN	Faruk Çetinkaya	04.10.24	04.10.24	100%
Implementing Neural Networks	Berkay Yalman	04.13.24	04..14.24	100%
Implementing k-fold cross valid.	All Members	04.15.24	04.15.24	100%
Implementing SVM	Faruk Çetinkaya	04.16.24	04.16.24	100%
k-NN plots and results	Faruk Çetinkaya	04.17.24	04.17.24	100%
SVM plots and results	Faruk Çetinkaya	04.17.24	04.17.24	100%
Neural Networks plots and result	Berkay Yalman	04.17.24	04.17.24	100%
Writing project phase 1 report	All Members	04.17.24	04.21.24	100%
Improving Methods	All Members	05.05.24	05.11.24	100%
Comparision of Methods	All Members	05.12.24	05.12.24	100%
Writing Final Report	All Members	05.11.24	05.12.24	100%

Figure 10: Task Distribution and Implementation dates

Implemented Enhancements

Under the same train and test set, the methods were compared with each other and the best method for solving the classification problem were determined. More combinations were tried for feature engineering in the SVM algorithm but notably better combinations were not found. These feature engineering applications were also applied to other algorithms.

Anticipated and Encountered Challanges

The first challenge we faced was that the dataset was not balanced. We solved this problem by deleting a certain part of the data. The second challenge was that the data was not standardized. This was a problem for algorithms that involve distance calculation such as KNN. We standardized the data. Another general difficulty we had was that the execution time of the algorithms was too long. For example, the distance calculation in the k-NN algorithm took a lot of time. We solved this problem by fully vectorizing k-NN algorithm. SVM has not yet been vectorized yet, so it still runs slowly compared to other algorithms. Since the execution time is long, it was a bit difficult to find the ideal regularization parameter and learning rate in SVM.

Conclusion

In this project, we have a supervised binary classification problem. First of all, the data was analyzed and some pre-processing was applied. Then SVM, Neural Network, and k-NN algorithms were used to solve the problem. The best parameters of each algorithm were found using k-fold cross-validation. After the parameters were optimized, the algorithms were compared with each other, and the best-performing algorithm in terms of both efficiency and accuracy was the Neural Network.

REFERENCES

[1] “Machine learning: A probabilistic perspective" by K. Murphy
[2] https://en.wikipedia.org/wiki/Neural_network

[3] Lecture Slide 8

[4] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

[5] https://medium.com/@quantclubiitkgp/time-series-classification-using-dynamic-time-warping-k-nearest-neighbour-e683896e0861

[6] https://en.wikipedia.org/wiki/Support_vector_machine

[7] https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

[8] https://datatron.com/what-is-a-support-vector-machine/

[9] https://en.wikipedia.org/wiki/Cross-validation_(statistics)

[10] https://www.researchgate.net/figure/k-fold-cross-validation-process_fig4_361110608

Appendix

```
#IMPORTING SOME LIBRARIES

import pandas as pd
import numpy as np
import seaborn as sns
import math
import seaborn as sns
import matplotlib.pyplot as plt
import time as tm


#DATA INTILIZATION FOR K-NN AND SVM

data1 = pd.read_csv('magic04.csv', header=None)
data = np.array(data1)
X_gammas = data[0:12332,0:11]
index = np.arange(0, 12332-6688)
X_gammas=np.delete(X_gammas,index,axis=0)#excess gamma data deleted
X_hadrons = data[12332:19020,0:11]
data_balanced = np.vstack((X_gammas,X_hadrons))#balanced data
np.random.shuffle(data_balanced)#shuffle data
X = data_balanced[:,0:10]
X = X.astype(float)
X_mean = np.mean(X, axis=0)
X_std = np.std(X, axis=0)
X_standardized = (X - X_mean) /  X_std #Standartized data
Y = np.reshape(data_balanced[:,10], (13376,1))
Y = np.where(Y == 'h', 0, 1) # 0 for hadron, 1 for gamma




# CODE FOR CORRELATION BETWEEN FEATURES AND LABELS

y_mean = np.mean(Y)
scores = {}
predictor_labels =
['fLength','fWidth','fSize','fConc','fConc1','fAsym','fM3Long','fM3Trans','fAlpha','fDist']
for i in range(10):
    x_i = X_standardized[:, i]
    x_mean = np.mean(x_i)

    numerator = np.sum(np.multiply((x_i - x_mean) , (Y.flatten() - y_mean)))
    denominator = np.sqrt(np.sum((x_i - x_mean) ** 2) * np.sum((Y.flatten() - y_mean) ** 2))

    scores[f'Correlation between label and feature {predictor_labels[i]}'] = abs(numerator / denominator)
```

```
sorted_feature_ranking = sorted(scores.items(), key=lambda item: item[1], reverse=True)
sorted_feature_ranking


#CODE FOR KNN ALGORITHM IMPLEMENTATION

def knn(X_train, y_train, X_test, k):
    diff = X_train - X_test #find the differences between x_instance and X_train datas
    distances = np.sqrt(np.sum(diff ** 2, axis=1))
    k_indices = np.argsort(distances)[:k] #choose the first smallest k distance
    k_nearest_labels = y_train[k_indices] # find labels(0-1) for chosen indices

    gamma_probability = np.mean(k_nearest_labels) # probability of being class 1(gamma)

    return gamma_probability


# K-FOLD CROSS VALIDATION FOR K-NN

def k_fold_cross_validation(X, y, k, num_folds, threshold):
    # k= nearest number of neighbor, num_folds= k_fold number
    fold_length = len(X) // num_folds
    accuracies = []
    all_y_pred = []
    all_accuracy=[]
    TP = TN = FP = FN = 0
    for i in range(num_folds):
        y_true = []
        start_row = i * fold_length
        end_row = (i + 1) * fold_length
        X_test = X[start_row:end_row]
        y_test = y[start_row:end_row]
        X_train = np.concatenate((X[:start_row], X[end_row:]))
        y_train = np.concatenate((y[:start_row], y[end_row:]))
        y_true.extend(y_test)
        y_pred = []
        for x_test_instance in X_test:
            prediction = knn(X_train, y_train, x_test_instance, k)
            if prediction >=threshold:#according to threshold give 1 or 0 to prediction
                prediction=1
            else:
                prediction=0
            y_pred.append(prediction)
        accuracy=0
        tp = tn = fp = fn = 0
        for true, pred in zip(y_true, y_pred):
            if true == 1 and pred == 1:
                tp += 1
                TP += 1
            elif true == 0 and pred == 0:
                tn += 1
                TN += 1
            elif true == 0 and pred == 1:
                fp += 1
                FP += 1
            elif true == 1 and pred == 0:
                fn += 1
                FN += 1
        accuracy= 100*(tp+tn)/(tp+tn+fp+fn)#accuracy for each fold
        all_accuracy.append(accuracy) #all accuracies(mean will be desired result)
        conf_matrix = [[TN//100,FP//100], [FN//100, TP//100]]

    return  np.mean(all_accuracy),conf_matrix


# RUN K-NN ALGORITHM WITH BEST PARAMETERS FOUND

start_time = tm.time()
accurcy,all_confusion_matrix=k_fold_cross_validation(X_standardized, Y, 10, 6,0.55)

print(accurcy)
end_time = tm.time()
print("Execution Time: {:.2f}s".format(end_time - start_time))

#VISUALIZATION OF CONFUSION MATRIX FOR K-NN

plt.figure(figsize=(10, 7))
sns.heatmap(all_confusion_matrix, annot=True, fmt="d", cmap='Blues', cbar=False)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# FINDING BEST NUMBER OF NEIGHBOR(k)

accuracies=[]
confusion_matrices=[]
for k in range(2,20):
    accuracy,confusion_matrix=k_fold_cross_validation(X_standardized, Y, k, 6,0.6)
    accuracies.append(accuracy)

# THE PLOT OF K VS ACCURACY

plt.plot([2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],accuracies)
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.title('Number of Neighbors vs Accuracy')
plt.show()

# FINDING BEST PROBABILTY THRESHOLD

accuracies2=[]
confusion_matrices2=[]
for i in np.arange(0.1, 1, 0.05):
    accuracy2,confusion_matrix2=k_fold_cross_validation(X_standardized, Y, 10, 6, i)
    accuracies2.append(accuracy2)

# THE PLOT OF THRESHOLD VS ACCURACY

plt.plot([0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.7,0.75,0.8,0.85,0.9,0.95],accuracies2)
plt.xlabel('threshold')
plt.ylabel('Accuracy')
plt.title('Threshold vs Accuracy')
plt.show()

#CODE FOR SVM ALGORITHM IMPLEMENTATION

def SVM(learning_rate, lambd, n_iterations, X_train, y_train):
    w = np.zeros(X_train.shape[1]) # initialize weight vector
    b = 0
    y_modified = np.where(y_train <= 0, -1, 1) #change labels 0 to -1, 1 is still 1

    for _ in range(n_iterations):#update weight vector and b
        for i, x in enumerate(X_train):

            margin = y_modified[i] * (np.dot(x, w) - b)>= 1 #this are the derivative of cost function
            if margin:
                w -= learning_rate*(2*lambd*w)
            else:
                w -= learning_rate*(2*lambd*w-x*y_modified[i])
                b -= learning_rate*y_modified[i]
    return w,b


# K-FOLD CROSS VALIDATION FOR SVM

def k_fold_cross_validation_SVM(X, y,
num_folds,regularization_param,learning_rate,num_of_iterations):
    fold_size = len(X) // num_folds
    accuracies = []
    all_y_pred = []
    all_y_true = []

    for i in range(num_folds):
        start_index = i * fold_size
        end_index = (i + 1) * fold_size if i != num_folds - 1 else len(X)
        X_test = X[start_index:end_index]
        y_test = y[start_index:end_index]
        X_train = np.concatenate((X[:start_index], X[end_index:]))
        y_train = np.concatenate((y[:start_index], y[end_index:]))

        y_pred = []
        w,b= SVM(learning_rate, regularization_param, num_of_iterations,X_train,y_train)
        for x_test_instance in X_test:
            predict = np.sign(np.dot(x_test_instance, w) - b)
            if predict ==-1:
                prediction=0
            else:
                prediction=1

            y_pred.append(prediction)

        all_y_pred.extend(y_pred)
        all_y_true.extend(y_test)

    tp = tn = fp = fn = 0
    for true, pred in zip(all_y_true, all_y_pred):
        if true == 1 and pred == 1:
            tp += 1

        if true == 0 and pred == 0:
            tn += 1

        if true == 0 and pred == 1:
            fp += 1

        if true == 1 and pred == 0:
            fn += 1

    accuracy= 100*(tp+tn)/(tp+tn+fp+fn)
    conf_matrix = [[tn//num_folds, fp//num_folds], [fn//num_folds, tp//num_folds]]
    return conf_matrix, accuracy

# RUN SVM ALGORITHM WITH BEST PARAMETERS FOUND

start_time = tm.time()
confsion_matrix_SVM,accuracy_SVM= k_fold_cross_validation_SVM(X_standardized, Y,
10,0.0001,0.0001,50)

print(accuracy_SVM)
end_time = tm.time()
print("Execution Time: {:.2f}s".format(end_time - start_time))

#VISUALIZATION OF CONFUSION MATRIX FOR SVM

plt.figure(figsize=(10, 7))
sns.heatmap(confsion_matrix_SVM, annot=True, fmt="d", cmap='Blues', cbar=False)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

#APPLY FEATURE ENGINEERING TO FEATURE VECTOR

X_standardized1=np.hstack((X_standardized,X_standardized[:,1:2]*X_standardized[:,1:2]))
X_standardized2=np.hstack((X_standardized,X_standardized[:,1:2]*X_standardized[:,2:3]))
X_standardized3=np.hstack((X_standardized1,X_standardized[:,3:4]*X_standardized[:,2:3]))
X_standardized4=np.hstack((X_standardized,X_standardized[:,8:9]*X_standardized[:,2:3]))
X_standardized5=np.hstack((X_standardized,X_standardized[:,0:1]*X_standardized[:,0:1]))
```

```
X_standardized6=np.hstack((X_standardized3,X_standardized[:,1:2]*X_standardized[:,2:3]))
X_standardized7=np.hstack((X_standardized3,X_standardized[:,7:8]*X_standardized[:,8:9]))
X_standardized8=np.hstack((X_standardized6,X_standardized[:,4:5]*X_standardized[:,8:9]))
X_standardized9=np.hstack((X_standardized8,X_standardized[:,0:1]*X_standardized[:,8:9]))
X_standardized10=np.hstack((X_standardized8,X_standardized[:,8:9]*X_standardized[:,8:9]))

#  RUN FOR SEVERAL UPDATED FEATURE VECTORS

confsion_matrix_SVM2,accuracy_SVM2= k_fold_cross_validation_SVM(X_standardized1, Y,
10,0.0001,0.0001,50)
print(accuracy_SVM2)

confsion_matrix_SVM3,accuracy_SVM3= k_fold_cross_validation_SVM(X_standardized3, Y,
10,0.0001,0.0001,50)
print(accuracy_SVM3)

confsion_matrix_SVM5,accuracy_SVM5= k_fold_cross_validation_SVM(X_standardized6, Y,
10,0.0001,0.0001,50)
print(accuracy_SVM5)




#FINDING BEST REGULARIZATION PARAMETER FOR SVM

accuracies3=[]
confusion_matrices3=[]
values = [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1]

for lambd in values:
    confusion_matrix,accuracy=k_fold_cross_validation_SVM(X_standardized, Y, 3,lambd,0.0001,50)
    accuracies3.append(accuracy)

#THE PLOT OF ACCURACY VS REGULARIZATION CONSTANT

plt.plot(values,accuracies3,marker='o')
plt.xscale('log')
plt.xlabel('lambda')
plt.ylabel('Accuracy')
plt.title('Regularization constant vs Accuracy')
plt.show()

#FINDING BEST LEARNING RATE FOR SVM

accuracies4=[]
confusion_matrices4=[]
values2 = [0.000001, 0.00001, 0.00003, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01]

for i in values2:
    confusion_matrix,accuracy=k_fold_cross_validation_SVM(X_standardized, Y, 3,0.0001,i,50)
    accuracies4.append(accuracy)
```

```
#THE PLOT OF ACCURACY VS LEARNING RATE FOR SVM

plt.plot(values2,accuracies4,marker='o')
plt.xscale('log')
plt.xlabel('learning rate')
plt.ylabel('Accuracy')
plt.title('Learning Rate vs Accuracy')
plt.show()

#FINDING IDEAL NUMBER OF ITERATION FOR SVM

accuracies5=[]
confusion_matrices5=[]
values3 = [int(10), int(20), int(30), int(40), int(50), int(60), int(70), int(80),
int(90),int(100),int(150),int(200)]

for i in values3:
    confusion_matrix,accuracy=k_fold_cross_validation_SVM(X_standardized, Y, 3,0.0001,0.0001,i)
    accuracies5.append(accuracy)


#THE PLOT OF ACCURACY VS NUMBER OF ITERATIONS FOR SVM

plt.plot(values3,accuracies5,marker='o')

plt.xlabel('num_of_iterations')
plt.ylabel('Accuracy')
plt.title('num_of_iterations vs Accuracy')
plt.show()




#PCA

Xn_T = Xn.T

CM_r= np.dot(Xn_T,Xn) #Covariance Matrix

CM = CM_r * (1/Xn.shape[0])

eigenvalues, eigenvectors = np.linalg.eig(CM)

sort_indices = np.argsort(eigenvalues)

sort_indices = sort_indices[::-1]

percentage_array = np.array(np.zeros([1,10]))

cumulative_percentage_array = np.array(np.zeros([1,10]))

total_variation = np.sum(eigenvalues)

pc = 0
```

```
for i in sort_indices:
    percentage_array[0,pc] =( eigenvalues[i] / total_variation ) * 100.0
    pc += 1
explained = 0
pc = 0
for i in sort_indices:
    explained += eigenvalues[i]
    cumulative_percentage_array[0,pc] =( explained / total_variation ) * 100.0
    pc += 1
pc_label = np.array([1,2,3,4,5,6,7,8,9,10])
plt.bar(pc_label.T,np.array(percentage_array[0,:]))
plt.xlabel('Principal Components')
plt.ylabel('Variance Explained')
plt.title('Variances Explained by Principal Components')
plt.show()
pc_label = np.array([1,2,3,4,5,6,7,8,9,10])
plt.bar(pc_label.T,np.array(cumulative_percentage_array[0,:]))
plt.xlabel('Principal Components')
plt.ylabel('Total Variance Explained')
plt.title('Cumulatively Variances Explained by Principal Components')
plt.show()

first_pc = eigenvectors[:,sort_indices[0]]
second_pc = eigenvectors[:,sort_indices[1]]
third_pc = eigenvectors[:,sort_indices[2]]

projection_matrix = np.array([first_pc,second_pc])
projection_matrix_3d = np.array([first_pc,second_pc,third_pc])
projections_of_properties = np.array(np.zeros([10,2]))
projections_of_properties = np.dot(projection_matrix,np.eye(10))
```

```
n = 0

arrows = []

for projection in projections_of_properties.T:

    arrow = plt.arrow(0, 0, projection[0], projection[1], head_width=0.03, head_length=0.03,
fc=predictor_colors[n], ec=predictor_colors[n])

    arrows.append(arrow)

    n += 1


plt.xlim(-1, 1)

plt.ylim(-1, 1)

plt.xlabel('First Principal Component')

plt.ylabel('Second Principal Component')

plt.title('Properties Projected onto first two principal components')


plt.grid(True)

plt.gca().set_aspect('equal', adjustable='box')

plt.legend(arrows,predictor_labels, fontsize = 'x-small')

plt.savefig('projection_of_properties.png',dpi = 300)

plt.show()


X2 = data.iloc[5644:,:].values

np.random.shuffle(X2)


k = 1000

X2 = X2[:k,:]

X2p = X2[:,:10]

X2l = X2[:,10:]

X2p = X2p.astype(float)


x_m = np.mean(X2p, axis = 0)

x_s = np.std(X2p, axis = 0)

X2p = (X2p - x_m)/x_s
```

```
sample_points = []
sample_names = []
red_added = False
blue_added = False
projections_of_data_instances = np.dot(projection_matrix_3d,X2p.T).T

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for i in range(k):
    if X2l[i] == 'g':
        point = ax.scatter(projections_of_data_instances[i,0],projections_of_data_instances[i,1],
projections_of_data_instances[i,2], color = 'red',alpha = 1.0, s = 1)
        if not red_added:
            sample_points.append(point)
            sample_names.append('g')
            red_added = True
    else:
        point = ax.scatter(projections_of_data_instances[i,0],projections_of_data_instances[i,1],
projections_of_data_instances[i,2], color = 'blue', alpha = 1.0, s = 1)
        if not blue_added:
            sample_points.append(point)
            sample_names.append('h')
            blue_added = True
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.title('Data instances Projected onto first two principal components')
plt.gca().set_aspect('equal', adjustable='box')
plt.legend(sample_points,sample_names,fontsize = 'x-small')

plt.savefig('projection_of_samples.png',dpi = 300)
plt.show()


#DATA INTIALIZATION FOR NEURAL NETWORKS AND PCA

data = pd.read_csv('magic04.csv',sep=',', header=None,
names=['fLength','fWidth','fSize','fConc','fConc1','fAsym','fM3Long','fM3Trans','fAlpha','fDist','class'])


# print(data.head())

predictor_labels =
['fLength','fWidth','fSize','fConc','fConc1','fAsym','fM3Long','fM3Trans','fAlpha','fDist']

predictor_colors = np.array(['red', 'blue', 'green', 'orange', 'purple', 'cyan', 'magenta', 'yellow', 'black',
'gray'])

#Center the Data

raw_data = data.iloc[5644:,:].values

np.random.shuffle(raw_data)

X = raw_data[:,:10]

X = X.astype(float)

raw_label = raw_data[:,10:]

Y = np.where(raw_label == 'g',1,0)


x_mean = np.mean(X, axis = 0)

x_std = np.std(X, axis = 0)


X = (X - x_mean)/x_std

#TESTING HYPERPARAMETERS

test_start = 1

test_end = 50

test_step = 1

test_name = ""

test_x = ""

test_y = ""

test_array = np.zeros((int((test_end-test_start)/test_step)+1,1))

x_array = []
```

```
test_i = 0
for test_v in range(test_start, test_end + test_step, test_step):
    x_array.append(test_v)
    #NEURAL NETWORK
    ild = 10 #Input layer dimension
    sld = 30 #Second layer dimension
    tld = 30 #Third layer dimension
    old = 1 #Output layer dimension
    #k-fold cross validation parameters
    k = 100 #Fold number
    l = 40 #Training session count over folds
    m = X.shape[0]
    fold_length = int(m/k)
    spare = m % fold_length
    X = X[:m-spare]
    Y = Y[:m-spare]
    X_folded = np.empty((k, fold_length, X.shape[1]), dtype=X.dtype)
    Y_folded = np.empty((k, fold_length, Y.shape[1]), dtype=Y.dtype)
    learning_rate = 2.0/k
    start_limit = 1.0
    slw = np.random.uniform(-start_limit,start_limit,(sld,ild)) #Input layer to second layer weights
    tlw = np.random.uniform(-start_limit,start_limit,(tld,sld)) #Second layer to third layer weihhts
    olw = np.random.uniform(-start_limit,start_limit,(old,tld)) #Third layer to output layer weights
    decision_boundary = 0.5
    g_slw = np.zeros(slw.shape)
    g_tlw = np.zeros(tlw.shape)
    g_olw = np.zeros(olw.shape)
    def relu(x):
        return np.maximum(x, 0)
    def relu_derivative(x):
        return np.where(x >= 0, 1, 0)
```

```
logistic_limit = 10.0
def logistic(v):
    v = np.clip(v,-logistic_limit, logistic_limit)
    return 1/(1+np.exp(-v))
def logistic_derivative(v):
    return logistic(v) * (1 - logistic(v))
def loss_function_derivative(y_label,y_pred):
    #Square loss function assumed
    return -2.0 * (y_label - y_pred)
def compute_prediction(input_vector):
    second_layer_values = relu(np.dot(slw,input_vector))
    third_layer_values = relu(np.dot(tlw,second_layer_values))
    output_layer_value = logistic(np.dot(olw,third_layer_values))
    return output_layer_value

def binary_prediction(input_vector):
    return (compute_prediction(input_vector) >= decision_boundary).astype(int)

def compute_batch_gradient(input_vector,y_label):
    # input_vector = input_vector.reshape(-1,1)
    v_sl = np.dot(slw,input_vector) #induced local field of second layer
    second_layer_values = relu(v_sl)
    v_tl = np.dot(tlw,second_layer_values) #induced local field of third layer
    third_layer_values = relu(v_tl)
    v_ol = np.dot(olw,third_layer_values) #induced local field of output layer
    y_pred = logistic(v_ol)
    d_ol = -loss_function_derivative(y_label, y_pred) * logistic_derivative(v_ol) #delta of output layer
    sg_olw =  np.dot(-d_ol, third_layer_values.T) #gradient for output layer weights
    d_tl = d_ol * np.multiply(olw.T,relu_derivative(v_tl)) #delta of third layer
    sg_tlw = np.dot(-d_tl,second_layer_values.T) # gradient for third layer weights
    d_sl = np.dot(tlw.T,d_tl)*relu_derivative(v_sl)
```

```
        sg_slw = np.dot(-d_sl, input_vector.T) #gradient for second layer weights
        return sg_olw,sg_tlw,sg_slw


    #K-FOLD CROSS VALIDATION
    for i in range(k):
        start_row = i * fold_length
        end_row = (i + 1) * fold_length
        X_folded[i] = X[start_row:end_row, :]
        Y_folded[i] = Y[start_row:end_row, :]
    avg_accuracy = 0
    avg_gradient_size = np.zeros((l,1))
    step = 0
    gradient_size = np.zeros((l,1))
    accuracy_over_train = np.zeros((l,1))
    CF = np.zeros((2,2))
    for i in range(l):
        for j in range(k): #Train Gradient descent
            if (i % k) == j:
                continue
            g_olw,g_tlw,g_slw = compute_batch_gradient(X_folded[j,:,:].T, Y_folded[j,:,:].T)
            olw = olw - learning_rate * g_olw
            tlw = tlw - learning_rate * g_tlw
            slw = slw - learning_rate * g_slw
            gradient_size[step] = gradient_size[step] + np.sum(np.abs(g_slw)) + np.sum(np.abs(g_olw)) +
np.sum(np.abs(g_tlw))
        step += 1


        y_pred = binary_prediction(X_folded[(i%k),:,:].T)
        y_label = Y_folded[(i % k),:,:].T


        #Confusion Matrix
        FN = (1-y_pred) * y_label
        TN = (1-y_pred) * (1 - y_label)
        FP = y_pred * (1 - y_label)
        TP = y_pred * y_label

        CF[0,0] += np.sum(TN)
        CF[0,1] += np.sum(FP)
        CF[1,0] += np.sum(FN)
        CF[1,1] += np.sum(TP)

        y_accuracy = y_pred == y_label
        accuracy = np.sum(y_accuracy)/y_accuracy.shape[1]
        accuracy_over_train[i] = accuracy
        avg_accuracy += accuracy * (1/l)
        avg_gradient_size += gradient_size * (1/l)
    plt.plot(avg_gradient_size)
    plt.show()
    plt.plot(accuracy_over_train)
    title = "Accuracy over k fold cross validation, avarage: {:.3f}".format(avg_accuracy)
    plt.title(title)
    plt.show()
    print(avg_accuracy)
    end_time = tm.time()
    print("Execution Time: {:.2f}s".format(end_time - start_time))
    test_array[test_i] = avg_accuracy
    test_i += 1
plt.plot(test_array)
plt.title(test_name)
plt.xlabel(test_x)
plt.ylabel(test_y)
plt.show()
```

CODE ADDED IN THE FINAL STAGE

#TRAIN-TEST SPLIT

```
def train_test_split(X, y, test_size):
    feature_eng1= np.hstack((X,X[:,1:2]*X[:,1:2]))
    feature_eng2= np.hstack((feature_eng1,X[:,3:4]*X[:,2:3]))
    feature_eng3= np.hstack((feature_eng2,X[:,1:2]*X[:,2:3]))

    num_test = int(len(X) * test_size)
    indices = np.random.permutation(len(X))

    test_indices = indices[:num_test]
    train_indices = indices[num_test:]

    X_train,X_test,y_train,y_test = X[train_indices],X[test_indices],y[train_indices],y[test_indices]
    X_mean = np.mean(X_train, axis=0)
    X_std = np.std(X_train, axis=0)
    X_train = (X_train - X_mean) /  X_std
    X_test  = (X_test - X_mean) /  X_std

    X_train2 = np.hstack((X_train,X_train[:,1:2]*X_train[:,1:2]))
    X_test2  = np.hstack((X_test,X_test[:,1:2]*X_test[:,1:2]))
    y_train2,y_test2 =  y[train_indices],y[test_indices]


    X_train3 = np.hstack((X_train2,X_train[:,3:4]*X_train[:,2:3]))
    X_test3  = np.hstack((X_test2,X_test[:,3:4]*X_test[:,2:3]))
    y_train3,y_test3 = y[train_indices],y[test_indices]

    X_train4 = np.hstack((X_train3,X_train[:,1:2]*X_train[:,2:3]))
    X_test4  = np.hstack((X_test3,X_test[:,1:2]*X_test[:,2:3]))
    y_train4,y_test4 = y[train_indices],y[test_indices]


    """X_train2,X_test2,y_train2,y_test2 =  feature_eng1[train_indices],
feature_eng1[test_indices],y[train_indices],y[test_indices]
    X_mean2 = np.mean(X_train2, axis=0)
    X_std2 = np.std(X_train2, axis=0)
    X_train2 = (X_train2 - X_mean2) /  X_std2
    X_test2 = (X_test2 - X_mean2) /  X_std2"""

    """"X_train3,X_test3,y_train3,y_test3 =
feature_eng2[train_indices],feature_eng2[test_indices],y[train_indices],y[test_indices]
    X_mean3 = np.mean(X_train3, axis=0)
    X_std3 = np.std(X_train3, axis=0)
    X_train3 = (X_train3 - X_mean3) /  X_std3
    X_test3 = (X_test3 - X_mean3) /    X_std3"""
    """"X_train4,X_test4,y_train4,y_test4 =
feature_eng3[train_indices],feature_eng3[test_indices],y[train_indices],y[test_indices]
    X_mean4 = np.mean(X_train4, axis=0)
    X_std4 = np.std(X_train4, axis=0)
    X_train4 = (X_train4 - X_mean4) /  X_std4
    X_test4 = (X_test4 - X_mean4) /  X_std4"""

    return X_train, X_test, y_train, y_test, X_train2,X_test2,y_train2,y_test2,
X_train3,X_test3,y_train3,y_test3,X_train4,X_test4,y_train4,y_test4


#TRAIN-TEST SPLIT_METHOD_KNN

def train_test_split_k_nearest(X_train, X_test, y_train, y_test, k, threshold):
    # k= nearest number of neighbor, num_folds= k_fold number
    all_accuracy=[]
    TP = TN = FP = FN = 0
    y_true = []
    y_pred = []
    y_true.extend(y_test)
    for x_test_instance in X_test:
        prediction = knn(X_train, y_train, x_test_instance, k)
        if prediction >=threshold:#according to threshold give 1 or 0 to prediction
            prediction=1
        else:
            prediction=0
        y_pred.append(prediction)
    accuracy=0
    tp = tn = fp = fn = 0
    for true, pred in zip(y_true, y_pred):
        if true == 1 and pred == 1:
            tp += 1
            TP += 1
        elif true == 0 and pred == 0:
            tn += 1
            TN += 1
        elif true == 0 and pred == 1:
            fp += 1
            FP += 1
        elif true == 1 and pred == 0:
            fn += 1
            FN += 1
    accuracy= 100*(tp+tn)/(tp+tn+fp+fn)#accuracy for each fold
    conf_matrix = [[TN//100,FP//100], [FN//100, TP//100]]

    return  np.mean(accuracy),conf_matrix



#TRAIN-TEST SPLIT_METHOD_SVM
```

```
def train_test_split__SVM(X_train, X_test, y_train,
y_test,regularization_param,learning_rate,num_of_iterations):

    all_y_pred = []
    all_y_true = []
    all_y_true.extend(y_test)
    y_pred = []
    w,b= SVM(learning_rate, regularization_param, num_of_iterations,X_train,y_train)
    for x_test_instance in X_test:
        predict = np.sign(np.dot(x_test_instance, w) - b)
        if predict ==-1:
            prediction=0
        else:
            prediction=1
        y_pred.append(prediction)

    all_y_pred.extend(y_pred)


    tp = tn = fp = fn = 0
    for true, pred in zip(all_y_true, all_y_pred):
        if true == 1 and pred == 1:
            tp += 1

        if true == 0 and pred == 0:
            tn += 1

        if true == 0 and pred == 1:
            fp += 1

        if true == 1 and pred == 0:
            fn += 1

    accuracy= 100*(tp+tn)/(tp+tn+fp+fn)
    conf_matrix = [[tn, fp], [fn, tp]]
    return conf_matrix, accuracy


#TRAIN-TEST SPLIT_METHOD_NN

def test_neural_network(feature_train, feature_test, label_train, label_test):

    ild = feature_train.shape[1] #Input layer dimension
    sld = 30 #Second layer dimension
    tld = 30 #Third layer dimension
    old = 1 #Output layer dimension

    k = 100 #Batch Size
    l = 20 #Training session over train data

    m = feature_train.shape[0]
    batch_length = int(m/k)
    spare = m % batch_length

    feauture_train = feature_train[:m-spare]
    label_train = label_train[:m-spare]

    feauture_train_batched = np.empty((k, batch_length, feauture_train.shape[1]),
dtype=feauture_train.dtype)
    label_train_batched = np.empty((k, batch_length, label_test.shape[1]), dtype=label_train.dtype)

    learning_rate = 1.0/k

    start_limit = 0.1

    slw = np.random.uniform(-start_limit,start_limit,(sld,ild)) #Input layer to second layer weights
    tlw = np.random.uniform(-start_limit,start_limit,(tld,sld)) #Second layer to third layer weihhts
    olw = np.random.uniform(-start_limit,start_limit,(old,tld)) #Third layer to output layer weights

    decision_boundary = 0.5

    g_slw = np.zeros(slw.shape)
    g_tlw = np.zeros(tlw.shape)
    g_olw = np.zeros(olw.shape)

    def relu(x):
        return np.maximum(x, 0)

    def relu_derivative(x):
        return np.where(x >= 0, 1, 0)

    logistic_limit = 10.0

    def logistic(v):
        v = np.clip(v,-logistic_limit, logistic_limit)
        return 1/(1+np.exp(-v))

    def logistic_derivative(v):
        return logistic(v) * (1 - logistic(v))

    def loss_function_derivative(y_label,y_pred):
        #Square loss function assumed
        return -2.0 * (y_label - y_pred)

    def compute_prediction(input_vector):
        second_layer_values = relu(np.dot(slw,input_vector))
        third_layer_values = relu(np.dot(tlw,second_layer_values))
        output_layer_value = logistic(np.dot(olw,third_layer_values))
        return output_layer_value
```

```
def binary_prediction(input_vector):
    return (compute_prediction(input_vector) >= decision_boundary).astype(int)


def compute_batch_gradient(input_vector,y_label):
    # input_vector = input_vector.reshape(-1,1)
    v_sl = np.dot(slw,input_vector) #induced local field of second layer
    second_layer_values = relu(v_sl)
    v_tl = np.dot(tlw,second_layer_values) #induced local field of third layer
    third_layer_values = relu(v_tl)
    v_ol = np.dot(olw,third_layer_values) #induced local field of output layer
    y_pred = logistic(v_ol)

    d_ol = -loss_function_derivative(y_label, y_pred) * logistic_derivative(v_ol) #delta of output layer
    sg_olw =  np.dot(-d_ol, third_layer_values.T) #gradient for output layer weights

    d_tl = d_ol * np.multiply(olw.T,relu_derivative(v_tl)) #delta of third layer
    sg_tlw = np.dot(-d_tl,second_layer_values.T) # gradient for third layer weights

    d_sl = np.dot(tlw.T,d_tl)*relu_derivative(v_sl)
    sg_slw = np.dot(-d_sl, input_vector.T) #gradient for second layer weights

    return sg_olw,sg_tlw,sg_slw

#Slice the data into mini batches
for i in range(k):
    start_row = i * batch_length
    end_row = (i + 1) * batch_length
    feauture_train_batched[i] = feature_train[start_row:end_row, :]
    label_train_batched[i] = label_train[start_row:end_row, :]

CF = np.zeros((2,2)) #Confusion Matrix

for n in range(l):
    for j in range(k): #Train Gradient descent
        g_olw,g_tlw,g_slw = compute_batch_gradient(feauture_train_batched[j,:,:].T,
label_train_batched[j,:,:].T)
        olw = olw - learning_rate * g_olw
        tlw = tlw - learning_rate * g_tlw
        slw = slw - learning_rate * g_slw

y_pred = binary_prediction(feature_test.T)
y_label =label_test.T

#Confusion Matrix
FN = (1-y_pred) * y_label
TN = (1-y_pred) * (1 - y_label)
FP = y_pred * (1 - y_label)
TP = y_pred * y_label

CF[0,0] += np.sum(TN)
```

```
    CF[0,1] += np.sum(FP)
    CF[1,0] += np.sum(FN)
    CF[1,1] += np.sum(TP)

    y_accuracy = y_pred == y_label
    accuracy = 100*np.sum(y_accuracy)/y_accuracy.shape[1]

    return accuracy

#INITIALIZE THE DATA

X_train, X_test, y_train, y_test, X_train2,X_test2,y_train2,y_test2,
X_train3,X_test3,y_train3,y_test3,X_train4,X_test4,y_train4,y_test4= train_test_split(X, Y, 0.2)


#ACCURACY RESULTS AND PERFORMANCE COMPARISON

start_time1 = tm.time()
nn_accuracy_normal_data = test_neural_network(X_train, X_test, y_train, y_test)
end_time1 = tm.time()
nn_execution_time1 = end_time1 - start_time1

start_time2 = tm.time()
nn_accuracy_feature_engineered_1 = test_neural_network(X_train2, X_test2, y_train2, y_test2)
end_time2 = tm.time()
nn_execution_time2 = end_time2 - start_time2

start_time3 = tm.time()
nn_accuracy_feature_engineered_2 = test_neural_network(X_train3, X_test3, y_train3, y_test3)
end_time3 = tm.time()
nn_execution_time3 = end_time3 - start_time3


start_time4 = tm.time()
nn_accuracy_feature_engineered_3 = test_neural_network(X_train4, X_test4, y_train4, y_test4)
end_time4 = tm.time()
nn_execution_time4 = end_time4 - start_time4


start_time1 = tm.time()
confsion_matrix_SVM,accuracy_SVM= train_test_split__SVM(X_train, X_test, y_train,
y_test,0.0001,0.0001,50)
end_time1 = tm.time()
svm_execution_time1 = end_time1 - start_time1


start_time2 = tm.time()
confsion_matrix_SVM2,accuracy_SVM2= train_test_split__SVM(X_train2, X_test2, y_train2,
y_test2,0.0001,0.0001,50)
end_time2 = tm.time()
svm_execution_time2 = end_time2 - start_time2
```

```
start_time3 = tm.time()
confsion_matrix_SVM3,accuracy_SVM3= train_test_split__SVM(X_train3, X_test3, y_train3,
y_test3,0.0001,0.0001,50)
end_time3 = tm.time()
svm_execution_time3 = end_time3 - start_time3


start_time4 = tm.time()
confsion_matrix_SVM6,accuracy_SVM6= train_test_split__SVM(X_train4, X_test4, y_train4,
y_test4,0.0001,0.0001,50)
end_time4 = tm.time()
svm_execution_time4 = end_time4 - start_time4


start_time1 = tm.time()
accurcy1,all_confusion_matrix=train_test_split_k_nearest(X_train, X_test, y_train, y_test,6,0.55)
end_time1 = tm.time()
knn_execution_time1 = end_time1 - start_time1


start_time2 = tm.time()
accurcy2,all_confusion_matrix=train_test_split_k_nearest(X_train2, X_test2, y_train2, y_test2,6,0.55)
end_time2 = tm.time()
knn_execution_time2 = end_time2 - start_time2


start_time3 = tm.time()
accurcy3,all_confusion_matrix=train_test_split_k_nearest(X_train3, X_test3, y_train3, y_test3,6,0.55)
end_time3 = tm.time()
knn_execution_time3 = end_time3 - start_time3


start_time4 = tm.time()
accurcy4,all_confusion_matrix=train_test_split_k_nearest(X_train4, X_test4, y_train4, y_test4,6,0.55)
end_time4 = tm.time()
knn_execution_time4 = end_time4 - start_time4


import numpy as np
import matplotlib.pyplot as plt

nn_accuracy = [nn_accuracy_normal_data, nn_accuracy_feature_engineered_1,
nn_accuracy_feature_engineered_2, nn_accuracy_feature_engineered_3]
nn_execution_time = [nn_execution_time1, nn_execution_time2, nn_execution_time3,
nn_execution_time4]

svm_accuracy = [accuracy_SVM, accuracy_SVM2, accuracy_SVM3, accuracy_SVM6]
svm_execution_time = [svm_execution_time1, svm_execution_time2, svm_execution_time3,
svm_execution_time4]

knn_accuracy = [accurcy1, accurcy2, accurcy3, accurcy4]
knn_execution_time = [knn_execution_time1, knn_execution_time2, knn_execution_time3,
knn_execution_time4]
```

```
x_label = ["Normal Data", "Feature Engineering 1", "Feature Engineering 2", "Feature Engineering 3"]

legend_labels = ["Neural Network", "Support Vector Machine", "k-Nearest Neighbours"]

# Plotting Accuracy Comparison of Models
plt.figure(figsize=(15, 10))
plt.plot(range(1, 5), nn_accuracy, 'o-', label='Neural Network')
plt.plot(range(1, 5), svm_accuracy, 'o-', label='Support Vector Machine')
plt.plot(range(1, 5), knn_accuracy, 'o-', label='k-Nearest Neighbours')
plt.xlim(0.5, 4.5)
plt.ylim(74, 88)
plt.xticks(range(1, 5), x_label)
plt.xlabel('Data Processing')
plt.ylabel('Accuracy(%)')
plt.title('Accuracy Comparison of Models')
plt.legend()


# Plotting Execution Time Comparison of Models
plt.figure(figsize=(15, 10))
plt.plot(range(1, 5), nn_execution_time, 'o-', label='Neural Network')
plt.plot(range(1, 5), svm_execution_time, 'o-', label='Support Vector Machine')
plt.plot(range(1, 5), knn_execution_time, 'o-', label='k-Nearest Neighbours')
plt.xlim(0.5, 4.5)
plt.ylim(0, 5)
plt.xticks(range(1, 5), x_label)
plt.xlabel('Data Processing')
plt.ylabel('Execution Time(s)')
plt.title('Execution Time Comparison of Models')
plt.legend()
```