EEE 485 Statistical Learning and Data Analytics Project First Report

Fuat Arslan **Melih Berk Yılmaz**

1. Problem

Classification is an important task, which is encountered in almost all fields. In this project, we strive to accomplish galactical object classification using tabular data retrieved from Sloan Sky Survey SDSS-DR16. There are 3 types of objects, in our case, Galaxy, Star and Quasi-Stellar Object (QSO).

2. Data

Data retrieved from a Kaggle named "Sloan Sky Survey SDSS-DR16" dataset whose link as follows: (www.kaggle.com/datasets/rockdeldiablo/sloandigital-sky-survey-dr16-70k?resource=download).
Dataset is prone to classification. There are 3 classes, GALAXY, STAR, QSO. In total there are 70,000 samples. Each sample has 17 features. However, the dataset is a bit imbalanced. From GALAXY, 49,690 samples (71%); from STAR class, 13,494 samples (19%) and finally, from QSO, only 6,816 samples (10%) exist. Features and their meanings listed as follows:

Objid: Unique SDSS identifier
RA: Right Ascension
Dec: Declination
psfMag_u: PSF (Point Spread Function) flux in the u band
psfMag_g: PSF flux in the g band
psfMag_r: PSF flux in the r band
psfMag_i: PSF flux in the i band
psfMag_z: PSF flux in the z band
run: identifies the specific scan
rerun: A reprocessing of an imaging run
camcol: identify the scanline within the run
Field: Field number
class: object class (galaxy, star or quasar object)
redshift: the Redshift of the object
plate: ID of the plate used for the telescope at the time the image was taken
mjd: modified Julian Date, i.e. the date at which the data were taken
fiberid: fiber ID

class feature is separated from overall data as labels. ***objid*** is unique to every sample of the data. Therefore, considering that it will not add any information, we removed this feature completely. ***rerun*** feature is constant for each sample in the data, thus again, we removed this feature by taking into account that it will not add any information.

3. Preprocess:

Preprocessing is a very crucial part of most machine learning projects. Thus, we did not use raw data directly; instead, we applied a couple of algorithms to increase the effectiveness of the ML algorithms. It could be verified from the GitHub [1] page; we separated these data processing into two. The first one is named preprocessing again, and the second is feature engineering.

Preprocessing: It consists of Pandas Numpy Converter, Normalizer, MinMax Scaling, Encoder and Outlier Removal.

➢ ***Pandas Numpy Converter:*** We designed all of the structure works on numpy ndarrays. However, data is retrieved as pandas and turned to numpy and we did not want to lose column name information. Therefore this converter was used for conversation.

➢ ***Normalizer:*** It is also called standardization. This method makes data have 0 mean and 1 variance. It normalizes every feature in the dataset by calculating their mean and subtracting from each element of the feature. Then divide by standard deviation of the feature. It is proven that this technique sometimes helps the model learn underlying distribution, prevents saturation, gets rid of highly imbalanced data values like one is 0.1, other one is 100, and most importantly accelerates the learning [2,3].

➢ ***MinMax Scaling:*** It is the way of mapping feature values to a specific interval. It is stated that this scaling is essential when a distance-based model is used like KNN. As aforementioned, It prevents imbalanced data values and abrupt distances [N1].

- ➢ ***Encoder:*** Some data features can be categorical, which cannot be processed by ML algorithms. Therefore, they should have been turned to numerical values. Our implementation has two options, one is the label encoder which gives a number to each unique categorical value of a feature and keeps the meaning of these numerical values with respect to categorical value. The second option is one-hot, which is, most of the time, a better choice to prevent giving a comparable value. It creates a one-hot representation of the features. Since our only categorical feature labels, we used this method to create a one-hot or value assignment. For the neural network, we chose one-hot; for KNN numeric values.
- ➢ ***Outlier Removal:*** Before moving on to the machine learning algorithm, the data must be cleaned from outlier samples since they can negatively contribute to the decision making system of the ML algorithm. In this step, we decided to apply z-score based outlier removal. To do this, the data is standardized by subtracting the mean and dividing by the standard deviation. then , the values in the range [-3*std, 3*std]. Statistically, the values between absolute of 3*std consist of 99.7% of the data, the values above which is called outlier [4]. As seen from below figures, the psf_Mag_u feature has some samples at -9999 (it cannot be seen since there are 10-20 samples).

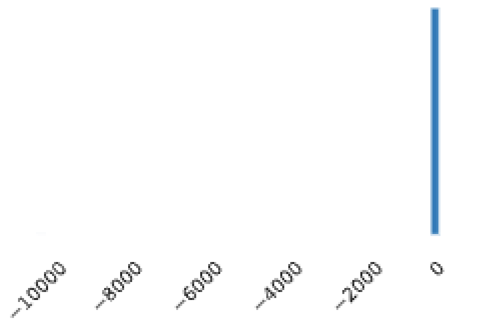


Figure 1: psf_Mag_u Feature Plot with Original Dataset

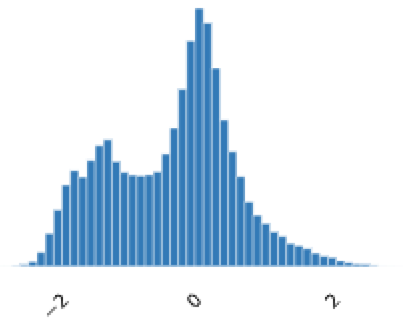
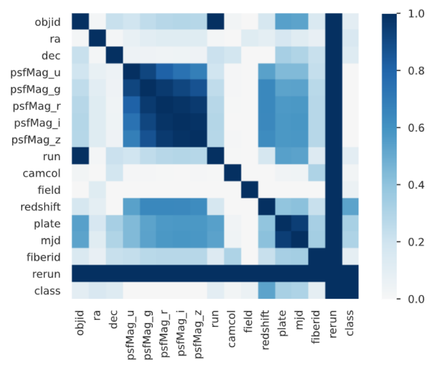


Figure 2: psf_Mag_u Feature Plot After Outlier Removal

Feature Engineering: This part includes Correlation, Principal Component Analysis (PCA), and Backward Elimination.

- ➢ ***Correlation:*** Correlation matrix shows the linear correlation of features with other features and the label. It cannot measure the nonlinear correlation, however, linear correlations can also cause gradient descent algorithm to slow down. Therefore, we investigated the linear correlation of the features with itself and label.



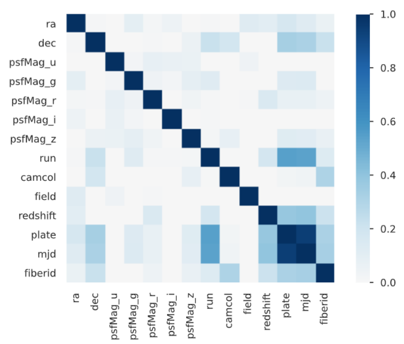


Figure3 (Left): Correlation Matrix of Original Dataset
Figure 4 (Right): Correlation Matrix After PCA

Figure 1 demonstrates the correlation of features including labels. As it is seen from the above figure, some of the features are highly correlated with each other. So, decorrelating them before applying a ML model might increase the efficiency and the result.

➢ ***Principal Component Analysis (PCA):*** PCA is a dimensionality reduction and decorrelation algorithm [5]. According to the information theory, information is embedded in the variance [6]. Therefore, the PCA algorithm tries to find the direction of the highest variance in the given dataset. Then, it finds the second highest variance direction which is orthogonal to the first one. By adding new orthogonal vectors to the previous ones, it projects the given data matrix into a new vector space whose basis is the set of orthogonal vectors. If the data is projected into a new space with lower dimension, it is called dimensionality reduction. If the dimension is kept the same, then, it is called decorrelation. Since data has 15 features, using PCA for dimensionality reduction is not beneficial in our case since we do not have a sparse feature set. We investigated the correlation matrix output. There is a high linear correlation between some of the features as it can be seen from figure 2. The features of psfMag_u, psfMag_g, psfMag_r, psfMag_i and psfMag_z are highly correlated. Therefore, this part of the data is passed through the PCA algorithm for decorrelation so that they are linearly independent. As explained, firstly the data is standardized, which maps values into a range in our case [-3,3]. In this range, the values of psfMag_u, psfMag_g, psfMag_r, psfMag_i and psfMag_z are very close to each other. Therefore, PCA output yielded very small values. As a result, PCA output is linearly mapped into range [-3,3] again using MinMax scaling. Figure 3 demonstrates the decorrelated correlation matrix.

➢ ***Backward Elimination:*** This algorithm is beneficial to decrease the number of features. Firstly, the data is fit to any machine learning model and accuracy is measured, called benchmark accuracy. Then, one of the feature vectors is dropped from the data and new data is fit to the same model and accuracy for the new dataset is measured. This process is applied for all feature vectors. If dropping one of the features increases the benchmark accuracy, then this feature vector is dropped from the feature set to make the algorithm more efficient. This algorithm continues to implement the same steps for dropping the second feature and so on. If dropping any one of the features does not increase the accuracy, then the algorithm stops. Since there are 15 features, this algorithm is not applied for feature elimination. Instead, it is used to gain insight about the importance of each feature. But, it can be used for XGBoost in the final demo.

4. Model Selection

4.1. Neural Network: Neural Networks or Multi Layer Perceptron (MLP) is an algorithm inspired from the learning mechanism of the brain. The NN algorithm is useful in terms of feature engineering. It completes the feature extraction by itself in the hidden layers. Therefore, finding nonlinear relations from a feature set is accomplished automatically by the algorithm.

4.1.1. Model Structure

At the beginning of the algorithm, the parameters of the MLP layers are initialized randomly. Equating all parameters to zero at the first step is very inefficient; therefore, there are some approaches for good initialization. We included 3 of them, Normal, Xavier and He initialization.

- Normal initialization draws values from a Standard Normal Gaussian Distribution for each parameter
- Xavier initialization: $W_{mxn} = \sqrt{\frac{6}{m+n}}$
- He initialization: $he = \frac{2}{\sqrt{input\ dimension}}, W_i = N(output\ dim,\ input\ dim)\ *\ he$

After parameter initialization, firstly the input data is multiplied by a weight vector. Then, the resulting vector is passed through a nonlinear function such as ReLU, Sigmoid, Tanh to increase nonlinearity. This procedure is done as much as the number of hidden layers. Specifically for classification, the final result is

passed through the Softmax function to obtain the probability of which class the sample belongs to. This is called the forward pass.
Secondly, the output result is equated to the class with the highest probability. In this stage, the loss is computed using Cross Entropy Loss. This loss is used to optimize the parameters of the model by trying to minimize it.
Finally, the derivative of the cost with respect to each layer input, layer weight and layer bias is calculated and all parameters are updated. Formulas below show the mathematical operations explained.

$$CrossEntropyLoss = -\sum_{i} y_{i} * log(o_{i})\ Forward\ Pass\ Z = WX + Bias,\ Activation\ Pass\ A = \delta(Z)$$

$$Softmax\ \delta(A) = O = \frac{e^{x_i}}{\sum_{i=1}^{3} e^{x_i}},\ \frac{\partial CELoss}{\partial O} = O - Y,\ \frac{\partial CELoss}{\partial W_{last\ layer}} = \frac{\partial CELoss}{\partial O} * \frac{\partial O}{\partial W_{last\ layer}} = \frac{\partial CELoss}{\partial O} * A_{last\ layer}^{T},$$

$$\frac{\partial CELoss}{\partial Bias_{last\ layer}} = \frac{\partial CELoss}{\partial O} * \frac{\partial O}{\partial bias_{last\ layer}} = \sum \frac{\partial CELoss}{\partial O},\quad \frac{\partial CELoss}{\partial A_{last\ layer}} = \frac{\partial CELoss}{\partial O} * \frac{\partial O}{\partial A_{last\ layer}} = W_{last\ layer}^{T} \frac{\partial CELoss}{\partial O}$$

These formulas show how forward and backward propagation is completed for MLP. They are demonstrated for just one layer; however, they propagate through every hidden layer.

4.1.2. Optimizer

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla w_t$$
$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * (\nabla w_t)^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

Figure 5: Adam Optimizer Parameter Update Formula

After derivatives are calculated, these parameters need to be updated so that the cost is decreased. The gradient descent algorithm finds the direction of the highest change of the variable and goes in the reverse direction of it to decrease the loss. We used a more sophisticated optimization algorithm called Adaptive Moment Estimation (Adam) optimizer. It combines the notion of Adaptive Gradient Algorithm and Root Mean Square Propagation. It updates parameters using the exponential moving average and square of the gradients [7]. This algorithm is more efficient, faster and converges better.
Instead of using a stochastic method, we implemented a mini batch based optimization algorithm. Mini batch algorithm firstly randomly shuffles the whole data and parses data into batches with equal sizes. This process is done in each epoch. Therefore, in each epoch the data in each batch is reshuffled. This technique increases the generalization ability of the model and prevents overfitting since the data is shuffled in each epoch. It prevents the model from memorizing the data.

4.1.3. Results

Before implementing any model, Preprocessing and Feature Engineering parts are excellently completed. These parts are very important to obtain higher accuracy and better generalization. Then, we constructed a L layer neural network, however, we used 2 layer MLP since the dimension of the data is small. The structure of the MLP:

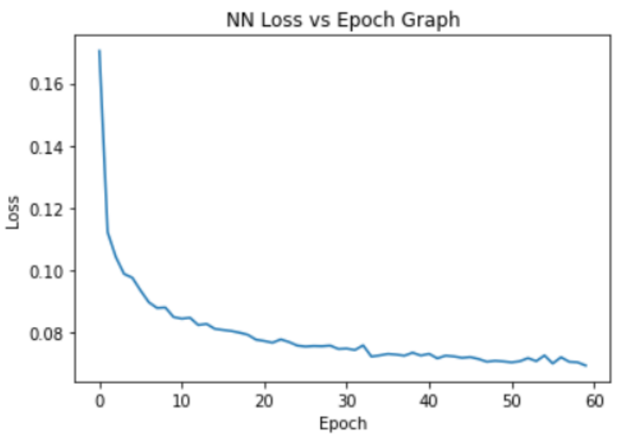
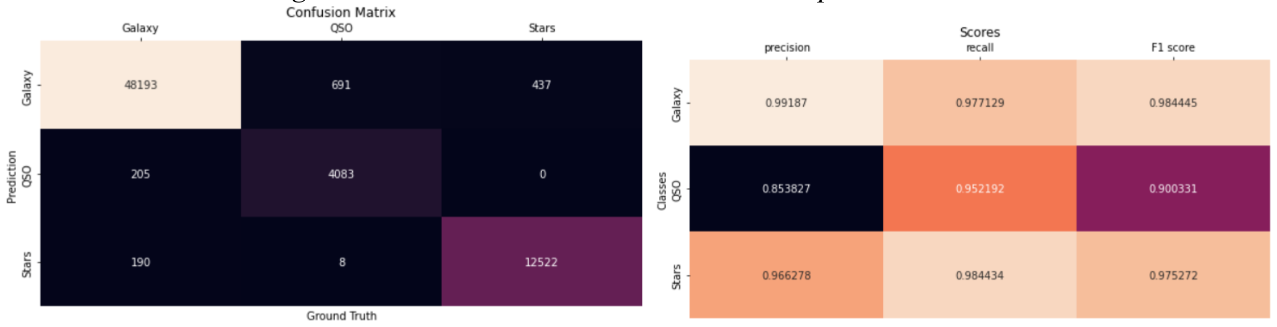


Figure 6: Training Loss for Neural Network

- ***First hidden layer:*** *32 Neurons, ReLU activation function, Xavier Initialization*
- ***Second hidden layer:*** *8 Neurons, ReLU activation function, Xavier Initialization*
- ***Output layer:*** *3 Neurons, Softmax activation Function, Xavier Initialization*

- ***Model Parameters:*** *Adam Optimizer, Mini Batch Size = 64, Epoch = 60, Loss = Cross Entropy Loss, Learning Rate = 0.01, Beta1 = 0.9, Beta2 = 0.999, epsilon = 1e-8*


Confusion Matrix

Prediction \ Ground Truth	Galaxy	QSO	Stars
Galaxy	48193	691	437
QSO	205	4083	0
Stars	190	8	12522

Scores

Classes	precision	recall	F1 score
Galaxy	0.99187	0.977129	0.984445
QSO	0.853827	0.952192	0.900331
Stars	0.966278	0.984434	0.975272

Figure 7: Confusion Matrix and Precision, Recall, F1 Score Table

Figure 6 doesn't demonstrate all the details of the result. Low loss doesn't mean anything, especially for classification problems. Therefore, we calculated the confusion matrix, precision, recall and F1 score of the model. As seen from figure 7, the accuracy is 98%. Even though there is a high imbalance in the dataset, the model can predict rare classes with high precision and accuracy. Precision, Recall and F1 scores are excellent for each class.

In order to understand the generalization of the network, we need validation scores to see whether the model can truly predict the unseen data. Thus, 3 fold Cross Validation is used.

Accuracy of fold 1 : % 97.7566
Accuracy of fold 2 : % 97.6932
Accuracy of fold 3 : % 97.2817
Average loss: 0.0872

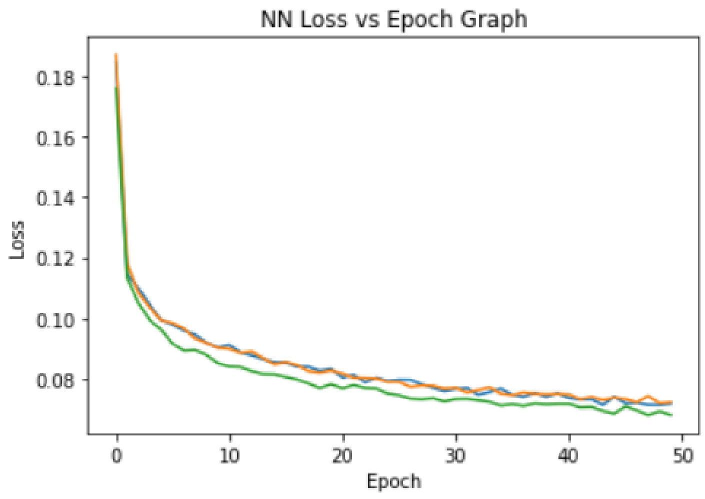


Figure 8: Neural Network Loss Plot for 3 Fold Cross Validation

As seen from figure 8, the model generalizes the test set with a high accuracy level (almost 98%). 2 Layer MLP model with Adam optimizer can reach high accuracy levels with test set as well.

4.2. K-Nearest Neighbor: It is a non-parametric supervised classification algorithm. This algorithm does not involve the learning phase. When it comes to the prediction of a class, it measures the distance from training samples and chooses k nearest of them; then, according to neighbors labels, it predicts the output label. It is a simple algorithm however it is very slow and costly in terms of prediction[8, 9]. The overall algorithm is explained in the below figure.

There are some extensions of KNN. First, according to the distance metric. We implemented a basic euclidean metric. However, in the literature, there were possibilities like manhattan distance; fortunately, our code is open to this extension, and it will be implemented in the final. The second extension aspect was the searching algorithm. In the literature, there are tree-based methods like the KD tree or ball tree. We implemented the brute force method. These searching methods are not affecting the final outputs but only their speed. Thus, we are not considering adding these methods to the project. Thirdly, by the voting method. We implemented a weighted version of voting. In the basic version, only the frequency of a label in the neighborhood decides the label, but the weighted version considers the distance of the neighbor point. A vote is calculated by the inverse of the distance.

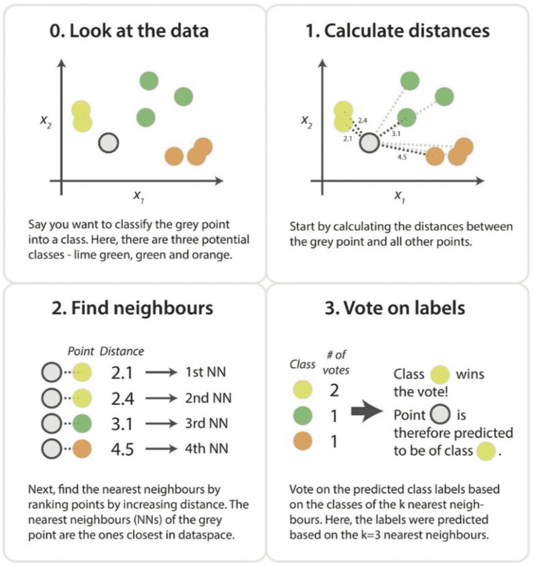

0. Look at the data
Say you want to classify the grey point into a class. Here, there are three potential classes - lime green, green and orange.
1. Calculate distances
Start by calculating the distances between the grey point and all other points.
2. Find neighbours
Point Distance
2.1 1st NN
2.4 2nd NN
3.1 3rd NN
4.5 4th NN
Next, find the nearest neighbours by ranking points by increasing distance. The nearest neighbours (NNs) of the grey point are the ones closest in dataspace.
3. Vote on labels
Class # of votes
Class wins the vote!
Point is therefore predicted to be of class .
Vote on the predicted class labels based on the classes of the k nearest neighbours. Here, the labels were predicted based on the k=3 nearest neighbours.

Figure 9: KNN Algorithm [8]

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

$$Manhattan: d(x, y) = \sum_{i=1}^{p} |x_i - y_i|$$

We implemented KNN and assessed its power by K-fold CV. Our K was 3. Therefore, the data is separated into 3 parts and spare 1 for validation. So the accuracy results of each fold as follows:

Accuracy of fold 1: 86.2590 %
Accuracy of fold 2: 88.8055 %
Accuracy of fold 3: 83.2195 %

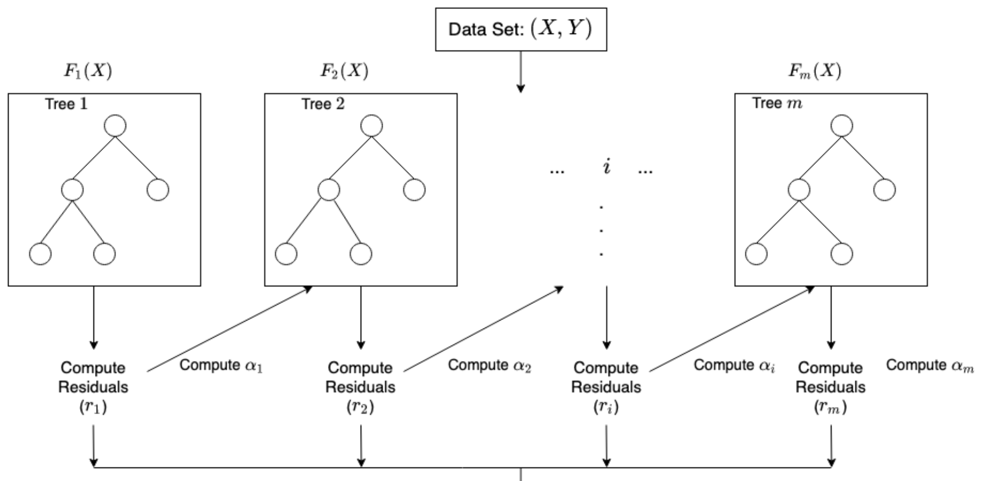
It seems that KNN is also very promising. However, it requires further tuning, which we could not do for the interim report. Also, we will further investigate this model in terms of precision, recall, and f1 score.

4.3. K-Means Clustering: It is an unsupervised clustering algorithm. It clusters data points by distances and finds a point that represents that class, called the centroid. The algorithm consists of three main steps. At the beginning, random centroids are chosen. This first centroid choice is further improved in the literature. We will soon mention what we implemented additionally. Firstly, data points are assigned to the nearest centroids in terms of a distance metric. Again this metric can change. We used euclidian. Then secondly, the mean of the data points belongs to a centroid calculated to use an iterated corresponding centroid. Thirdly iterate centroids and restart the algorithm and continue until it converges.

We used the K-means++ initialization technique to increase convergence speed and quality. This technique uses data points itself for initial centroids. Kmeans++ algorithm chooses one random point from the dataset and assigns it to a centroid. Then choose the next centroid as a data point that is furthest from other centroids [10].

We applied K-Mean to our data but did not work as expected. It could not separate Stars and QSO from Galaxy. After the discussion with our instructor, we came to the conclusion of using an unsupervised algorithm to supervised learning prone dataset. Therefore, we gave up using this algorithm. However, there is still a possibility to use this for feature selection or model selection. Thus we kept it inside our project library.

4.4. XGBoost Algorithm: It stands for Extreme Gradient Boosting. At the core, it is an efficient implementation of gradient boosted tree algorithm. This algorithm is again a supervised algorithm. It tries to predict accurately by using smaller and weaker models. XGBoos trains decision trees on a subset of the data then combines the result to get the prediction. Including regularizations makes it more generalizable than GBM. Then training is done by gradient optimization algorithms. The general algorithm is defined in the below figure.



$$F_m(X) \ = \ F_{m-1}(X) + \alpha_m h_m(X, r_{m-1}),$$

where α_i, and r_i are the regularization parameters and residuals computed with the i^{th} tree respectfully, and h_i is a function that is trained to predict residuals, r_i using X for the i^{th} tree. To compute α_i we use the residuals computed, r_i and compute the following: $arg \min_{\alpha} \ = \sum_{i=1}^{m} L(Y_i, F_{i-1}(X_i) + \alpha h_i(X_i, r_{i-1}))$ where $L(Y, F(X))$ is a differentiable loss function.

Figure 10: XGBoost Pseudo Algorithm [11]

We have not implemented XGBoost yet. Therefore, the algorithm review was kept a bit shallow.

5. Expected Challenges

XGBoost seems like a relatively hard algorithm. Therefore, we are expecting challenging times while implementing it. However, it will be a great opportunity to deep dive into a very popular and powerful algorithm.

We have made great progress in our project. We were expecting challenges with the imbalanced dataset. However, we successfully got it over with. Since we have already implemented two models, we more or less saw the problematic aspects and tackled them. Our results are very promising which reveals that we did something correctly. For instance to tackle imbalanced data we added a weighted voting version.

We were faced with code generalization and we are expecting to face it in the future as well. Some of our code requires some adjustments to work for each model. We are solving them whenever required. Thus it has never been a dead end and it will never be. Since we wrote very flexible code and completed the bulk of the work, we have already solved most of the problems. We are not expecting big challenges from now except XGBoost algorithm implementation.

6. Gantt Chart

Work Packages	Task Owner(s)	Nov 1-5	Nov 6-11	Nov 12-17	Nov 17-20
Literature Review for Algorithm Implementation	Fuat, Melih	🟪	🟪		
Preprocessing	Fuat	🟦			
Feature Engineering	Melih		🟧		
Neural Network	Fuat, Melih		⬛	⬛	⬛
K-Means Clustering	Fuat, Melih	🟫			
k-Nearest Neighbor	Fuat				🟩
XGBoost (Literature Review)	Fuat, Melih			🟨	🟨
Evaluator	Melih		🟦		

Figure 11: Gantt Chart for The Project Progress

References

[1]F. Arslan and M. Yılmaz, “ML from Strach,” *GitHub*. [Online]. Available: https://github.com/fuat-arslan/ML. [Accessed: 20-Nov-2022].

[2]S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” *arxiv*.

[3] A. Bhandari, “Feature scaling: Standardization vs normalization,” *Analytics Vidhya*, 15-Jun-2022. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/. [Accessed: 20-Nov-2022].

[4] A. P. Gulati, “Dealing with outliers using the Z-score method,” *Analytics Vidhya*, 02-Sep-2022. [Online]. Available: https://www.analyticsvidhya.com/blog/2022/08/dealing-with-outliers-using-the-z-score-method/. [Accessed: 20-Nov-2022].

[5]“Principal Component Analysis,” *Wikipedia*, 10-Nov-2022. [Online]. Available: https://en.wikipedia.org/wiki/Principal_component_analysis. [Accessed: 20-Nov-2022].

[6]W. R. Garner and W. J. McGill, “The Relation Between Information and Variance Analyses,” *Psychometrika*, vol. 21, no. 3, pp. 219–228, 1956.

[7]J. Brownlee, “Gentle Introduction To The Adam Optimization Algorithm For Deep Learning,” *MachineLearningMastery.com*, 12-Jan-2021. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. [Accessed: 20-Nov-2022].

[8]A. Christopher, “K-Nearest Neighbor,” Medium, 03-Feb-2021. [Online]. Available: https://medium.com/swlh/k-nearest-neighbor-ca2593d7a3c4. [Accessed: 20-Nov-2022].

[9]“K-Nearest Neighbors Algorithm,” *Wikipedia*, 10-Nov-2022. [Online]. Available: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm. [Accessed: 20-Nov-2022].

[10]“ML: K-means++ Algorithm,” *GeeksforGeeks*, 13-Jul-2021. [Online]. Available: https://www.geeksforgeeks.org/ml-k-means-algorithm/. [Accessed: 20-Nov-2022].

[11]D. Hudgeon and R. Nichol, “Machine Learning For Business: Using Amazon Sagemaker and Jupyter,” *Amazon*, 2020. [Online]. Available: https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost-HowItWorks.html. [Accessed: 20-Nov-2022].

CODE

All the codes are alos accessible from github : https://github.com/fuat-arslan/ML/

Metrics:

```
"""
This will include measurment metrics
"""
import copy as cp
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt


from utils import *

class kFold():
    def __init__(self):
        pass


    def eval(self,model,X, y, cost_metric, num_folds = 3,*argv):
        """
        real function to run CV alorithm
        """

        if cost_metric == 'CrossEntropy':
            cost = Cross_Entropy_Loss
        elif cost_metric == 'MSE':
            #mse = ((pred - y_val)**2).mean(axis=0)
            pass
        else:
            print('Enter a valid cost metric')

        total = 0

        seperators = [a for a in range(0,len(X),int(len(X)/num_folds))]
        X_copy = X.copy()
        y_copy = y.copy()
        for i in range(1,num_folds+1):

            model_copy = cp.deepcopy(model)
            X_val = X_copy[seperators[i-1]:seperators[i]].copy()
            y_val = y_copy[seperators[i-1]:seperators[i]]
            X_train =
np.delete(X_copy,range(seperators[i-1],seperators[i]),axis = 0)
            y_train =
np.delete(y_copy,range(seperators[i-1],seperators[i]),axis = 0)

            model_copy.learn(X_train,y_train,*argv)
            pred = model_copy.predict(X_val)

            loss = cost(pred, y_val.T)
            arg_pred = np.argmax(pred,axis=0)
            true_label = np.argmax(y_val,axis = 1)
            acc = np.sum(arg_pred == true_label)/arg_pred.shape[0]
            #print("Prediction shape: ", arg_pred.shape)
            print(f"Accuracy of fold {i} : %" ,np.round(100*acc,4))
            print('\n')

            total += loss/pred.shape[1]
        print("Average loss:", np.round(total/num_folds,4) )
        return total/num_folds


#Score generator function

class Evaluator():
    def __init__(self, prediction, true_value, label_dict, display = True):
        self.prediction = prediction
        self.true_value = true_value
        self.label_dict = label_dict
        self.display = display

        # sort label dictionary by value
        self.label_dict = dict(sorted(self.label_dict.items(), key=lambda
item: item[1]))


    def confusion_matrix(self, prediction, true_value, label_dict, display
= True):
        #number of classes
        num_class = np.unique(true_value).shape[0]
        confusion_mat = np.zeros((num_class, num_class))

        for i in range(prediction.shape[0]):
            confusion_mat[prediction[i], true_value[i]] += 1
        if display:
            result = sns.heatmap(confusion_mat, annot=True ,cbar = 
False,fmt='g')
            result.set(xlabel='Ground Truth', ylabel='Prediction',
                       xticklabels = list(label_dict.keys()),
                       yticklabels = list(label_dict.keys()))
            result.xaxis.tick_top()
            result.set_title("Confusion Matrix")
        return confusion_mat

    def scores(self):
        plt.figure(figsize = (10,10))
        plt.subplot(2,1,1)
        confusion_mat = self.confusion_matrix(self.prediction, 
self.true_value, self.label_dict, self.display)

        acc = np.trace(confusion_mat)/np.sum(confusion_mat)
        precision = np.diag(confusion_mat) / np.sum(confusion_mat, axis = 
0)
        recall = np.diag(confusion_mat) / np.sum(confusion_mat, axis = 1)
        f1_score = 2*precision * recall / (precision + recall)

        # recall, precision, f1 score corresponds to columns
        score = np.hstack((precision.reshape(-1,1), 
                           recall.reshape(-1,1), f1_score.reshape(-1,1)))

        plt.subplot(2,1,2)
        result = sns.heatmap(score, annot=True ,cbar = False,fmt='g')
        result.set(xticklabels=["precision", "recall", "F1 score"],
                   ylabel='Classes', yticklabels = 
list(self.label_dict.keys()))
        result.xaxis.tick_top()
        result.set_title("Scores")

        print("Accuracy: ", np.round(acc,2))
        return acc, score
```

KNN Model

```
import numpy as np
class KNN():
    def __init__(self, num_neig, metric ="euclidian" ,weighted =False):

        self.num_neig = num_neig
        self.metric = metric
        self.weighted = weighted

    def distance_metric(self, metric, mat1, mat2):
        if metric == "euclidian":
            return np.sqrt(np.sum((mat1 - mat2) ** 2, axis = 1))


    def learn(self, X, Y,):
        self.X = X
        self.Y = Y
        return self

    def predict(self, sample):
        eps = 1e-13
        predictions = []
        for j in range(len(sample)):
            sample_hat = np.tile(sample[j], (len(self.X), 1))
            #print(len(self.X))
            #print(sample_hat.shape)
            dist = self.distance_metric(self.metric, self.X, sample_hat)
            sorted_idx = np.argsort(dist)
            if self.weighted:
                u, indicies =
np.unique(self.Y[sorted_idx[:self.num_neig]],return_inverse=True)
                indexed_distances = dist[sorted_idx[:self.num_neig]]
                weighted_label = np.zeros(len(np.unique(self.Y)))
                for k in range(len(indicies)):
                    weighted_label[indicies[k]] +=
1/(indexed_distances[k]+eps)

                #print('predicition',u[np.argmax(weighted_label)])
                predictions.append(u[np.argmax(weighted_label)])
            else:

predictions.append(np.bincount(self.Y[sorted_idx[:self.num_neig]].astype(in
t).reshape(-1)).argmax())
```

```
            return predictions
```

K-Means

```
import numpy as np

class KMeans():
  def __init__(self, num_cluster, state, metric, seed = 1):
    self.num_cluster = num_cluster
    self.state = state
    self.seed = seed
    self.centroids = []
    self.metric = metric

  def initialize(self, min, max, data):
    if self.state == "random":
      np.random.seed(self.seed)
      self.centroids = [np.random.uniform(min,max) for _ in
range(self.num_cluster)]
      self.centroids = np.array(self.centroids)

    elif self.state == "data":
      idx = np.random.randint(0, data.shape[0], 3)
      self.centroids = data[idx]

    elif self.state == "k-means ++":
        idx = np.random.randint(0, data.shape[0])
        self.centroids.append(data[idx])
        self.centroids = np.array(self.centroids)
        print(self.centroids)

        for cent in range(self.num_cluster-1):
            print('cent is ', cent)
            min_distances=[]

            for point_idx in range(data.shape[0]):

                distances = []


                #look for predetermined centroids and find min distance
                #we determined cent+1 of centroid
                for pre_cid in range(cent+1):
                    #print(self.centroids[pre_cid].shape)
                    print(pre_cid)
                    dist =
self.distance_metric("euclidian",data[point_idx],self.centroids[pre_cid].re
shape((1,-1)))
                    distances.append(dist[0])

                # points possible minimum distances are stored.
                #print(distances)
                foo = np.min(distances)
                min_distances.append(foo)

            min_distances = np.array(min_distances)
            furthest_point_idx = np.argmax(min_distances)
            new_centroid = data[furthest_point_idx]
            print(new_centroid)
            self.centroids = np.append(self.centroids,
new_centroid.reshape((1,-1)),axis = 0)
            print('new cent')


        print("a")
    else:
        print('Wrong initialization Method')

  def distance_metric(self, metric, mat1, mat2):
    if metric == "euclidian":
      return np.sqrt(np.sum((mat1 - mat2) ** 2, axis = 1))

  def centroid_assignment(self, sample):
    sample_hat = np.tile(sample, (self.num_cluster, 1))
    distances = self.distance_metric(self.metric, sample_hat,
self.centroids)
    return np.argmin(distances)

  def learn(self, X, epoch = 100, tolerance = 1e-4):
    self.tolerance = self.num_cluster * tolerance
    self.min_, self.max_ = np.min(X, axis=0), np.max(X, axis=0)
    # random centroids are initialized
    self.initialize(self.min_, self.max_, X)

    iter, diff = 0, 0
    new_centroids = np.zeros((self.num_cluster, X.shape[1]))

    while iter < epoch and diff > self.tolerance:
        # cluster list will be used to keep data indexes of each cluster
        cluster_list = [[] for i in range(self.num_cluster)]
        for idx in range(X.shape[0]):
          cluster_id = self.centroid_assignment(X[idx])
          cluster_list[cluster_id].append(idx)

        # centroids update and tolerans calculation
        for i, cidx in enumerate(cluster_list):
          new_centroids[i] = np.mean(X[cidx], axis = 0)

        diff = np.sum(self.distance_metric(self.metric,
                                            self.centroids, new_centroids))
        self.centroids = new_centroids
        if iter % 10 == 0:
          print(iter)
        iter +=1

    return self

  def execute(self):
    pass

  def predict(self, X):
    label_list = []
    for idx in range(X.shape[0]):
      cluster_id = self.centroid_assignment(X[idx])
      label_list.append(cluster_id)
    return label_list

  def label_iden(self,X,y):
        label_map = {}
        # cluster list will be used to keep data indexes of each cluster
        cluster_list = [[] for i in range(self.num_cluster)]
        for idx in range(X.shape[0]):
            cluster_id = self.centroid_assignment(X[idx])
            cluster_list[cluster_id].append(idx)
        for i, cidx in enumerate(cluster_list):
            #taken from stackoverflow

#https://stackoverflow.com/questions/19909167/how-to-find-most-frequent-str
```

```
ing-element-in-numpy-ndarray
            unique,pos = np.unique(y[cidx],return_inverse=True) #Finds all
unique elements and their positions
            counts = np.bincount(pos)                     #Count the number
of each unique element
            maxpos = counts.argmax()
            label_map[str(i)] = unique[maxpos]
            #label_map[str(i)] = np.bincount(y[cidx]).argmax()
        return label_map

  def evaluate(self, predict, ground_truth):
        acc = np.sum(predict == ground_truth) / predict.shape[0]
        return acc
```

NEURAL NETWORKS

LAYERS

```
"""
This script includes neural network layers. For now there is just Dense
layer.
"""
import numpy as np
from utils import softmax, sigmoid, tanh, relu, initializer


class Dense():
    def __init__(self, input_dim, output_dim, method = "Xavier", activation
= "relu"):
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.activation = activation
        self.method = method
        self.weight = initializer(self.input_dim, self.output_dim,
self.method)
        self.bais = np.zeros((output_dim, 1))

    def update_params(self, w, b):
        self.weight = w.copy()
        self.bais = b.copy()
        return

    def forward(self, X):
        # linear forward
        self.X = X
        self.Z = self.weight @ self.X + self.bais

        # activation forward
        if self.activation == "softmax":
            A = softmax(self.Z)

        elif self.activation == "relu":
            A = relu(self.Z)

        elif self.activation == "sigmoid":
            A = sigmoid(self.Z)

        elif self.activation == "tanh":
            A = tanh(self.Z)

        return A

    def backward(self, dA):
        # derivative with respect to activation function
        if self.activation == "relu":
            dZ = relu(dA, self.Z)

        elif self.activation == "sigmoid":
            dZ = sigmoid(dA, self.Z)

        elif self.activation == "tanh":
            dZ = tanh(dA, self.Z)
        else:
            dZ = dA
        m = self.X.shape[1]
        # derivative with respect to weights, bais and X(current state)
        self.dweight = 1/m * np.matmul(dZ, self.X.T)
        self.dbais = 1/m * np.sum(dZ, axis = 1, keepdims=True)
        self.dX = np.matmul(self.weight.T, dZ)

        return self.dX
```

OPTIMIZER

"""
Optimizer scirpt. For now there is only adam optimizer.

```
"""

import numpy as np


class Optimizer():
    def __init__(self,method, learning_rate, beta, beta1, beta2, epsilon = 
1e-8):
        self.method = method
        self.learning_rate = learning_rate
        self.t = 1
        self.beta = beta
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.flag = True

    def initializer(self, layer_list):
        if self.method == "gd":
            pass
        if self.method == "sgd":
            pass
        if self.method == "momentum":
            pass
        if self.method == "adam":
            self.v = {}
            self.s = {}
            w = []
            b = []
            for layer in layer_list:
                w.append(np.zeros(layer.weight.shape))
                b.append(np.zeros(layer.bais.shape))
            self.v['W'] = w.copy()
            self.v['b'] = b.copy()
            self.s['W'] = w.copy()
            self.s['b'] = b.copy()


    def step(self, layer_list):
        if self.method == "gd":
            pass
        if self.method == "sgd":
            pass
        if self.method == "momentum":
            pass
        if self.method == "adam":
            if self.flag:
                self.initializer(layer_list)
                self.flag = False

            v_corrected = {'W':[],  'b':[]}
            s_corrected = {'W':[],  'b':[]}
            for i, layer in enumerate(layer_list):
                self.v['W'][i] = self.beta1 * self.v['W'][i] +
(1-self.beta1) * layer.dweight
                self.v['b'][i] = self.beta1 * self.v['b'][i] +
(1-self.beta1) * layer.dbais

                v_corrected['W'].append((self.v["W"][i] /
(1-self.beta1**self.t)))
                v_corrected['b'].append((self.v["b"][i] /
(1-self.beta1**self.t)))

                self.s['W'][i] = self.beta2 * self.s['W'][i] +
(1-self.beta2) * layer.dweight **2
                self.s['b'][i] = self.beta2 * self.s['b'][i] +
(1-self.beta2) * layer.dbais **2

                s_corrected['W'].append((self.s["W"][i] /
(1-self.beta2**self.t)))
                s_corrected['b'].append((self.s["b"][i] /
(1-self.beta2**self.t)))

                new_W = layer.weight -(self.learning_rate *
v_corrected['W'][i] / (s_corrected["W"][i]**(0.5) + self.epsilon))
                new_b = layer.bais -(self.learning_rate *
v_corrected['b'][i] / (s_corrected["b"][i]**(0.5) + self.epsilon))
                layer.update_params(new_W,new_b)

        self.t += 1
        return
```

Trainers

```
"""
This scirpt includes trainer of the network.
"""

from utils import mini_batch_generator, Cross_Entropy_Loss

import matplotlib.pyplot as plt

class NeuralNetwork():
    def __init__(self, layers, cost_function):
        self.layers = layers
        self.cost_function = cost_function

    def learn(self, X, Y, optimizer, max_epoch, batch_size):

        if self.cost_function == 'CrossEntopy':
            cost = Cross_Entropy_Loss

        else:
            print('invalid cost metric!')


        train_losses = []
        val_losses = []

        X = X.T
        Y = Y.T

        epoch = 0
        loss_list = []

        while epoch < max_epoch:
            loss = 0
            batches_list = mini_batch_generator(X, Y, batch_size)
            for batch in batches_list:
                A, label = batch
                for layer in self.layers:
                    A = layer.forward(A)

                loss += cost(A, label) / A.shape[1]

                dA = cost(A, label, back = True)
                for layer in self.layers[::-1]:
                    dA = layer.backward(dA)

                optimizer.step(self.layers)

            if epoch % 10 == 0:
                print ("Cost after epoch %i: %f" %(epoch,
loss/len(batches_list)))
            loss_list.append(loss/ len(batches_list))
            epoch += 1

        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.title("NN Loss vs Epoch Graph")
        plt.plot(loss_list)
        return

    def predict(self, data):
        A = data.T.copy()
        for layer in self.layers:
            A = layer.forward(A)

        return A
```

UTILS

```
import numpy as np
import matplotlib.pyplot as plt
import math
"""**utils**"""

def sigmoid(Z, dA = None):
    if not isinstance(dA, np.ndarray):
        A = 1/(1+np.exp(-Z))
    else:
        A = dA *  (1/(1+np.exp(-Z))) * (1- 1/(1+np.exp(-Z)))
    return A



def tanh(Z, dA = None):
    if not isinstance(dA, np.ndarray):
        A = np.tanh(Z)
    else:
        A = dA *  (1 - np.tanh(Z)**2)
    return A

def softmax(Z):
    num = np.exp(Z)
    denom = np.sum(num, axis = 0)
    A = num / denom
    return A

def relu(Z, dA = 0):
    if not isinstance(dA, np.ndarray):
        A = np.maximum(0, Z)
    else:
        dZ = np.array(Z, copy=True)
        dZ[dA <= 0] = 0
        A = dZ
    return A

def Cross_Entropy_Loss(O, Y,back = False):
    if not back:
        L = -np.sum(np.log(O) * Y)
    else:
        L = O - Y
    return L


#Network weight initilazier

def initializer(input_dim, output_dim, method):
    if method == "Xavier":
        w_i = (6 / (input_dim + output_dim)) ** 0.5
        weights = np.random.uniform(-w_i, w_i, size = (output_dim,
input_dim))

    elif method == "Normal":
        weights = np.random.normal(size = (output_dim, input_dim))

    elif method == "He":
        he = 2 / (input_dim) ** 0.5
        weights = np.random.rand(output_dim, input_dim) * he
    return weights
```

```
#Random mini batch generator

def mini_batch_generator(X, Y, batch_size):
    m = X.shape[1]
    mini_batches = []

    idx = list(np.random.permutation(m))
    shuffled_X = X[:, idx]
    shuffled_Y = Y[:, idx]

    full_batches = math.floor(m/batch_size)
    for k in range(0, full_batches):
        mini_batch_X = shuffled_X[:, k * batch_size : (k+1) * batch_size]
        mini_batch_Y = shuffled_Y[:, k * batch_size : (k+1) * batch_size]

        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    if m % batch_size != 0:
        mini_batch_X = shuffled_X[:, batch_size * math.floor(m /
batch_size) : ]
        mini_batch_Y = shuffled_Y[:, batch_size * math.floor(m /
batch_size) : ]

        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

PREPROCESS

```
import numpy as np
import time
import copy as cp

class PCA():
    def __init__(self):
        pass

    def learn(self,X,num_component=5):
        self.MEAN = np.mean(X, axis = 0)
        self.COV = np.cov((X - self.MEAN), rowvar = False)
        self.EIG_VAL , self.EIG_VEC = np.linalg.eigh(self.COV)
        self.num_component_ = num_component
        return self
    def execute(self,X):
        indexes = np.argsort(self.EIG_VAL)[::-1]
        sorted_eigenvalue = self.EIG_VAL[indexes]
        sorted_eigenvectors = self.EIG_VEC[:,indexes]

        eig_subset = sorted_eigenvectors[:,0:self.num_component_]
        X_reduced = (X - self.MEAN) @ eig_subset

        return X_reduced

    def fast(self,X,num_component=5):
        return self.learn(X,num_component).execute(X)


class Correlation():
    def __init__(self):
        pass

    def learn(self,X):
        pass

    def execute(self,X):
        return np.corrcoef(X,rowvar =False)

    def fast(self,X):
        return self.learn(X).execute(X)




 #Backward Elimination

class BackwardElimination():

    def __init__(self, num_elim, stop_cond, model, col_names):
        self.num_elim = num_elim
        self.stop_cond = stop_cond
        self.model = model
        self.col_names = col_names
    def learn(self, X, Y, *argv):

        self.del_col_idx = []
        # diff is initialized to make while conditioon true at the
beginning
        eliminated = 0
        diff = self.stop_cond + 1
        while eliminated < self.num_elim or self.stop_cond < diff:
            #benchmark model
            t1 = time.time()

            self.model.learn(X,Y,*argv)
            soft_out_bench= self.model.predict(X)

            pred_bench = np.argmax(soft_out_bench, axis = 0)
            true_label_bench = np.argmax(Y, axis = 1)


            bench_acc =
np.sum(pred_bench==true_label_bench)/pred_bench.shape[0]
            print("\n Bench Accuracy  : %" ,np.round(100*bench_acc,4))

            temp_acc_list = np.zeros((X.shape[1], 1))


self.model.layers[0].__init__(self.model.layers[0].input_dim-1,self.model.l
ayers[0].output_dim)
            for j in range(1,len(self.model.layers)):

self.model.layers[j].__init__(self.model.layers[j].input_dim,self.model.lay
ers[j].output_dim)
            opt = argv[0]
            opt.__init__(opt.method, opt.learning_rate, opt.beta,
opt.beta1,opt.beta2)
            #self.model.__init__(self.model.layers,self.model_out)
            #self.model = cp.deepcopy(self.model)


            for i in range(X.shape[1]):
                # fit model by droping one column for each column, get acc
                print(f"\n The feature of {self.col_names[i]} is removed")
                temp_data = np.delete(X, i, axis = 1)
                for j in range(0,len(self.model.layers)):

self.model.layers[j].__init__(self.model.layers[j].input_dim,self.model.lay
ers[j].output_dim)
                opt = argv[0]
                opt.__init__(opt.method, opt.learning_rate, opt.beta,
opt.beta1,opt.beta2)

                self.model.learn(temp_data, Y, *argv)
                soft_out = self.model.predict(temp_data)

                pred = np.argmax(soft_out, axis = 0)
                true_label = np.argmax(Y, axis = 1)


                temp_acc = np.sum(pred==true_label)/pred.shape[0]
                temp_acc_list[i]= temp_acc

                print("\n Accuracy  : %" ,np.round(100*temp_acc,4))
                print('\n')
            t2 = time.time()

            # get unuseful data and drop it
            idx_col = np.argmax(temp_acc_list)
            if temp_acc_list[idx_col] > bench_acc and
(temp_acc_list[idx_col] - bench_acc) > self.stop_cond:
                X = np.delete(X, idx_col, axis = 1)
                print("The column {} is dropped since accuray increased
from {} to {}".format(self.col_names[idx_col], np.round(100*bench_acc,4),
np.round(100*temp_acc_list[idx_col],4)))
                print("Elapsed time for droping {} column is {} mins {}
secs".format(self.col_names[idx_col], (t2-t1)//60, (t2-t1) - (t2-t1)//60 *
60))
                np.delete(self.col_names,idx_col)
                self.del_col_idx.append(idx_col)

                eliminated += 1
                diff = temp_acc_list[idx_col] - bench_acc

            else:
                print("Stopping condition has satisfied, no more
elimination")
                return self
```

```python
    def execute(self, X):
        X = np.delete(X, self.del_col_idx, axis = 1)
        return X

    def fast(self, X, Y, *argv):
        return self.learn(X, Y, *argv).execute(X)



"""
This scirpt includes function for preprocessing.
"""

import numpy as np
import pandas as pd
#Pandas Numpy converter

class pd_np_Converter():

    def __init__ (self,X,col = None):
        self.X = X
        if isinstance(X, pd.DataFrame):
            self.pandas_inst = True
            self.columns = X.columns
        else :
            self.pandas_inst = False
            self.columns = col

    def to_nup(self):
        if self.pandas_inst:
            return self.X.to_numpy()
        else:
            return self.X
    def to_pd(self):
        if self.pandas_inst:
            return self.X
        else:
            return pd.DataFrame(self.X,self.columns)


#normalization
```

```
class Normalizer():
    """
    It calculates the mean and standard deviation of the each future.
    It subrtacts the mean and divide by std to normalize.
    """
    def __init__ (self):
        pass

    def learn(self,X):
        self.MEAN = np.mean(X,axis=0) #Calculate the mean
        self.STD = np.std(X,axis=0) #Calculate the STD of each feature
        return self

    def execute(self,X):
        output = (X - self.MEAN)/self.STD
        return output

    def fast(self,X):
        return self.learn(X).execute(X)

#MinMax

class MinMax():
    """
    It maps the values to a ceratin interval.

    """
    def __init__(self,low = 0, high = 1):
        self.low_ = low
        self.high_ = high

    def learn(self,X):
        self.MIN = np.min(X,axis=0)
        self.MAX = np.max(X,axis=0)
        return self

    def execute(self,X):
        X_scaled = (X-self.MIN)/(self.MAX -self.MIN) #X features are mapped
(0,1)
        X_rescaled = X_scaled * (self.high_-self.low_) + self.low_
        return X_rescaled

    def fast(self,X):
        return self.learn(X).execute(X)


#one hot encoder

class Encoder():
    def __init__(self,one_hot = True):
        self.one_hot = one_hot

    def learn(self,X):
        self.label_map_list = {}
        self.num_features = X.shape[1]
        self.one_hot_list = {}
        self.categorical_idx = []

        X_copy = X.copy()
        for col_idx in range(self.num_features):
            col = X_copy[:,col_idx]


            if not np.issubdtype(type(col[0]), np.number):
                #print(type(col[0]))
                foo_map = {}
                self.categorical_idx.append(col_idx)

                u, indices = np.unique(col, return_index=True)

                for i, uniq in enumerate(u):
                    col[col == uniq] = i
                    foo_map[uniq] = i

                col = col.astype(int)
                if self.one_hot:
                    hot = np.zeros((col.size, col.max() + 1))
                    hot[np.arange(col.size), col] = 1
                else:
                    hot = col.reshape((-1,1))

                self.one_hot_list[str(col_idx)] = hot
                self.label_map_list[str(col_idx)] = foo_map
        return self

    def execute(self,X):
        for i in self.one_hot_list.values():
            X = np.concatenate((X, i),axis = 1)
            #print(X)

        X = np.delete(X,self.categorical_idx,1)

        return X.astype(float).astype(int)

    def fast(self,X):
        return self.learn(X).execute(X)

#Outlier Removal by Z score


class OutlierRemoval():
    def __init__(self, X, Y, threshold):
        self.X = X
        self.Y = Y
        self.threshold = threshold

    def learn(self, sdv_norm = False):

        if not sdv_norm:
            self.X = (self.X - np.mean(self.X, axis = 0)) / np.std(self.X,
axis = 0)

        return self

    def execute(self):
        data = np.hstack((self.X, self.Y))

        clean_data = data[np.all(abs(self.X) < self.threshold, axis=1)]
        self.outlier = data[~np.all(abs(self.X) < self.threshold, axis=1)]
        clean_X = clean_data[:,:-1]
        clean_Y = clean_data[:,-1]
        return clean_X.astype(float), clean_Y.reshape(-1,1)

    def fast(self, sdv_norm = False):
        return self.learn(sdv_norm).execute()
```