EEE 485 Statistical Learning and Data Analytics Project Final 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.
- ➢ ***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.
- ➢ ***Encoder:*** Some data features can be categorical, which cannot be processed by ML algorithms.We have implemented two types of encoders, one-hot and label encoders. We use the one-hot version for Neural Network and XGBoost and the label version for KNN.
- ➢ ***Outlier Removal:*** 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 [2]. 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).

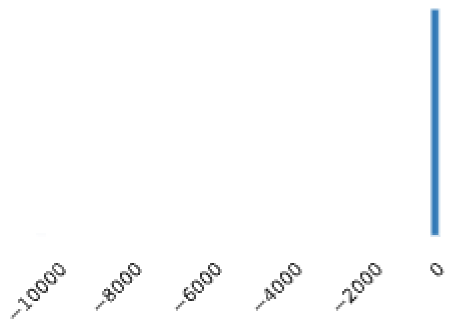


Figure 1: psf_Mag_u Feature Plot with Original Dataset

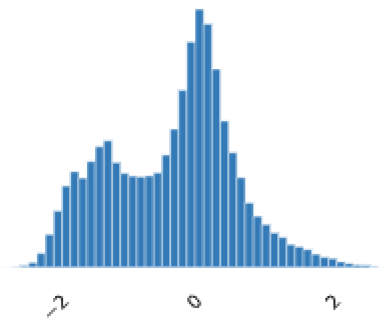
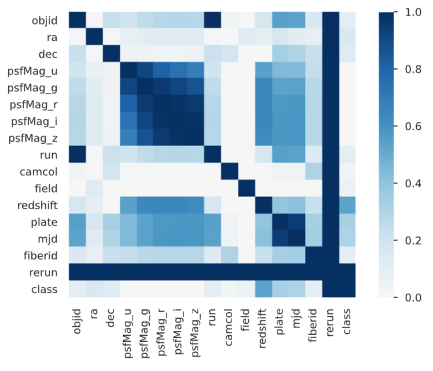


Figure 2: psf_Mag_u Feature Plot After Outlier Removal

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. Therefore, we investigated the linear correlation of the features with itself and label.



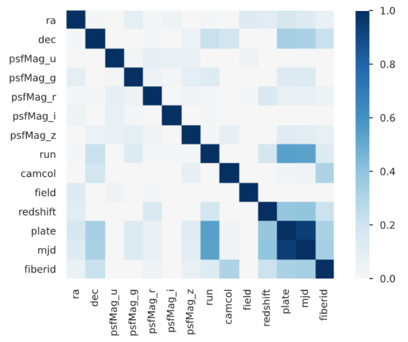


Figure3 (Left): Correlation Matrix of Original Dataset
Figure 4 (Right): Correlation Matrix After PCA

Figure 3 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 [3]. According to information theory, information is embedded in the variance [4]. 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. 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.

4. Model Selection

4.1. Neural Network: Neuraş Networks (NN) are very useful and robust models for both classification and regression tasks. The Neural Network algorithm is useful in terms of feature engineering. It completes the feature extraction by itself in the hidden layers. Using non-linear activation functions iteratively, it captures the nonlinear relations between the features, which is the general case in most real life datasets.
Moreover, NNs can capture hierarchical representations amongst the data. This is useful for machine learning algorithms to comprehend the complex patterns in the data. They are easy to work with high dimensional and sparse datasets. Since we have a dataset with dimensions 70,000x17, it is very efficient to use neural networks.

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.
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 that we have used 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 but 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 are updated to decrease the cost. 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 [5]. This algorithm is more efficient, faster and converges better.
Instead of using a stochastic method, we implemented a mini batch based optimization algorithm. 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. After preprocessing, we splitted the data into train, validation and test with 80%, 10%, 10% respectively. Since there is a high imbalance in the dataset, we used stratified split, which splits the data according to the percentages of each class.
There are lots of parameters in the neural network models. Parameters are the number of layers, the number of neurons in layers, activation function, learning rate, mini batch size, beta1, beta2, epsilon. Beta1, beta2 and epsilon are the parameters of adam optimizer. All the well-known neural network libraries, such as Pytorch, Keras, TensorFlow, set beta1=0.9, beta2=0.999, and epsilon=1e-8. Thus, we didn't optimize them since adding additional parameters increases optimization time exponentially. The remaining parameters are optimized using Grid Search with k-Fold, which is a very costly optimization technique.

Learning Rates = [0.0001, 0.001, 0.01]
Mini Batch Size = [16, 32, 64]
The Number of Layers and Neurons = [[14, 32, 8, 3], [14, 16, 32, 3], [14, 64, 32, 8, 3], [14, 16, 32, 64, 32, 16, 8, 3], [14, 32, 64, 128, 64, 32, 3], [14, 32, 32, 16, 3]]
Activation Function = ["relu", "sigmoid", "tanh"]
5 fold Cross Validation using 20 epochs

The Grid Search with 5 fold Cross Validation algorithm took 3.5 hours. Models are trained for 20 epochs and tested on the validation set. So, the k fold cv never sees the test data, which is used only after parameter tuning. We set the number of epochs to 20 for 5 fold cv since the model converges very fastly, it can reach more than 90% accuracy in 20 epochs. Moreover, there were 162 possibilities with the above grids, thus using more epochs results in time inefficiency. The below structures have the highest average accuracy rate on the validation set respectively out of 162 structures.

[learning_rate = 0.001, mini_batch_size = 16, layer_neuron = list([14, 32, 8, 3]), activation = 'tanh'], accuracy= 97.73%
[learning_rate = 0.01, mini_batch_size = 32, layer_neuron = list([14, 64, 32, 8, 3]), activation = 'sigmoid'], accuracy= 97.66%
[learning_rate = 0.001, mini_batch_size = 16, layer_neuron = list([14, 32, 8, 3]), activation = sigmoid], accuracy= 97.64%
[learning_rate = 0.001, mini_batch_size = 32, layer_neuron = list([14, 32, 8, 3]), activation = sigmoid], accuracy= 97.63%
[learning_rate = 0.01, mini_batch_size = 32, layer_neuron = list([14, 32, 8, 3]), activation = sigmoid], accuracy= 97.61%

As seen above, 4 of them have the structure of *list([14, 32, 8, 3]).* So, we didn't make a change on the best model. The model is trained with the best parameters.

- ***First hidden layer:*** *32 Neurons, tanh activation function, Xavier Initialization*
- ***Second hidden layer:*** *8 Neurons, tanh activation function, Xavier Initialization*
- ***Output layer:*** *3 Neurons, Softmax activation Function, Xavier Initialization*
- ***Model Parameters:*** *Adam Optimizer, Mini Batch Size = 16, Epoch = 90, Loss = Cross Entropy Loss, Learning Rate = 0.001, Beta1 = 0.9, Beta2 = 0.999, epsilon = 1e-8*

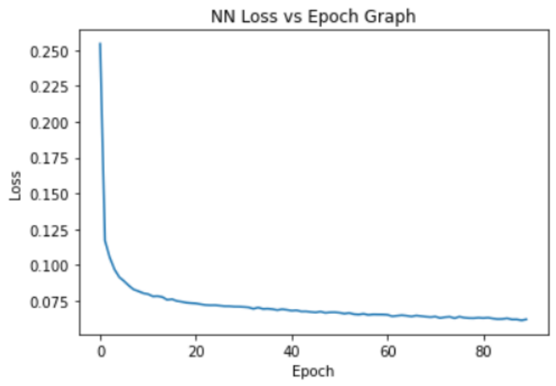
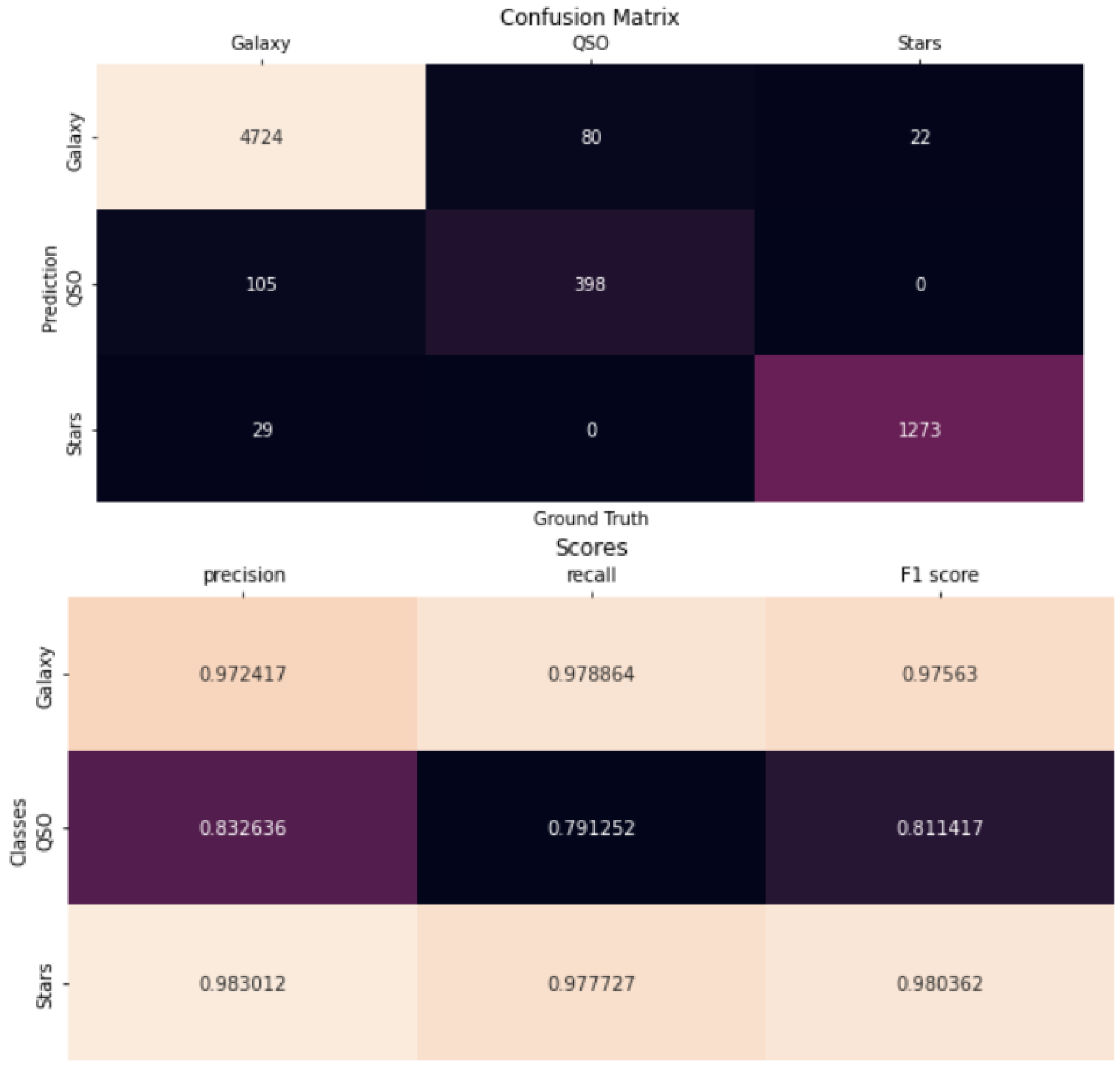


Figure 6: Training Loss for Neural Network

The training data reaches 98% accuracy rate with the given set up. Finally, the test set is passed through a trained optimal neural network model. The following figures show the confusion matrix, precision, recall and F1 score of the test results.



Confusion Matrix

Prediction \ Ground Truth	Galaxy	QSO	Stars
Galaxy	4724	80	22
QSO	105	398	0
Stars	29	0	1273

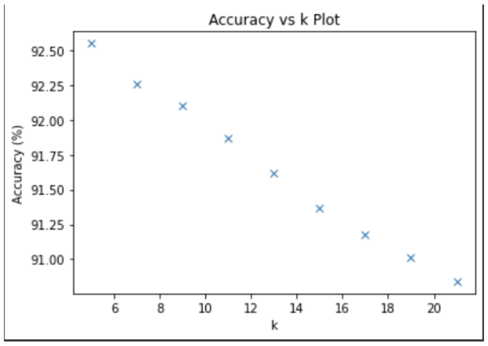
Scores

Classes	precision	recall	F1 score
Galaxy	0.972417	0.978864	0.97563
QSO	0.832636	0.791252	0.811417
Stars	0.983012	0.977727	0.980362

Figure 7: Confusion Matrix and Precision, Recall, F1 Score Table of the Best Neural Network Model

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 96%. 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. The scores are very close to 1, except for the class QSO, which is the 10% of the whole dataset. Maybe, increasing the number of QSO samples or adding a penalty term for not predicting QSO when the true label is QSO would increase its scores. However, we couldn't implement such an extra penalty term.

4.2. K-Nearest Neighbor: It is a non-parametric supervised classification algorithm. This algorithm does not involve the learning phase. It is a simple algorithm however it is very slow and costly in terms of prediction[6, 7].
There are some extensions of KNN. First, according to the distance metric. We implemented the euclidean metric. The second extension aspect was the search algorithm. In the literature, tree-based methods like the KD tree or ball tree are used. We implemented the brute force method. These searching methods are not affecting the final outputs but only their speed. Thus, we did not add 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.
Even though it is a simple algorithm, it works effectively. It does not require hyperparameter tuning except k and weighted voting. For KNN we do not need pre-knowledge about data and its distribution. Therefore, it is a perfect starting point. Its effectiveness increases with the dimension of the data. Furthermore, it is good to use noisy data. Thus, we chose KNN to start with the application because of its fast and effective features.
4.2.1 Results:
Splitted data explained in 4.1.3 was used to be consistent in terms of comparison and model evaluations. Meaning that, we used the same train, validation and test split for all algorithms in order to compare them at the end. KNN has two parameters that are optimized: the first one is the number of neighbors (K) and weighted voting.
We implemented KNN and tuned the parameters with K-fold where the fold number is 5. Therefore, the data is separated into 5 parts and spare 1 for validation. For each run accuracy on the validation part was calculated and the average of them gives the final accuracy result. This result score is used for parameter comparison. We searched `[5,7,9,11,13,15,17,19,21]` for K values and each of them checked with weighted voting and without it. Below two plots reveal the result of the parameter search. The left one shows experiments on the weighted voting version and the right side without weighted. Learning time takes 1.5 minutes, one 5 k-fold CV takes 11.5 minutes thus total takes 1.5 hours to complete 9 different K values searches. It was done for both weighted and unweighted versions thus the total is 3h.



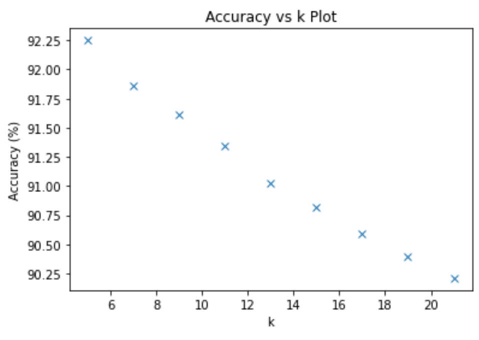
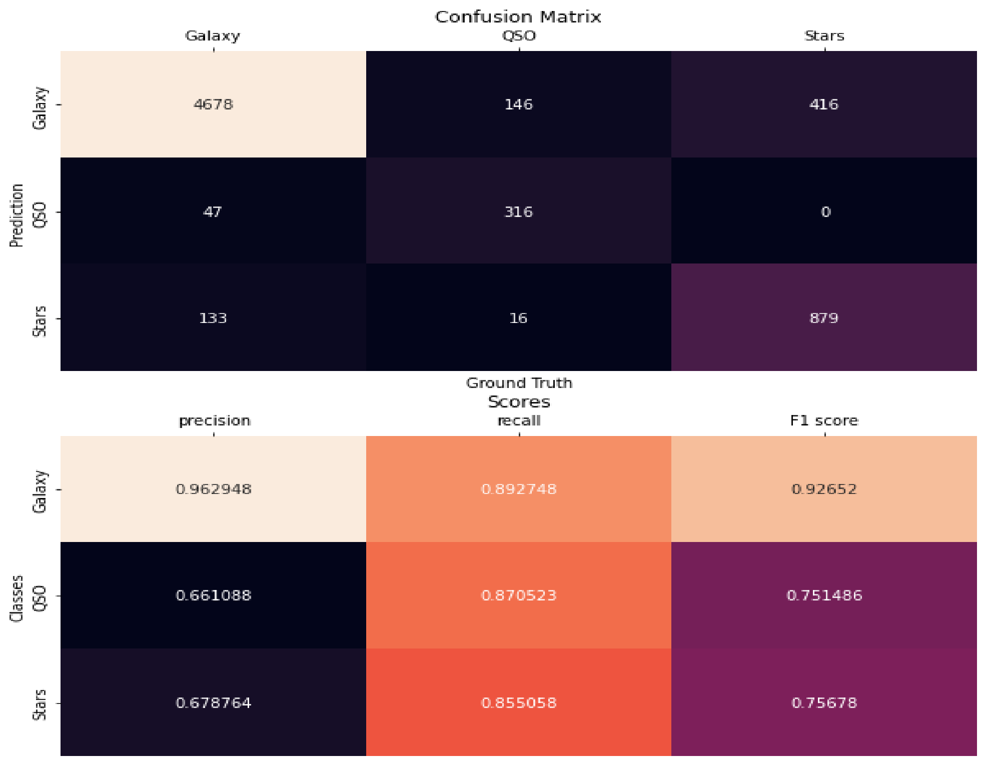


Figure 8 (Left): Weighted version K search Accuracy Plot
Figure 9 (Right): Unweighted version K search Accuracy Plot

As seen from the above figures 8-9 that for the weighted version there are no big differences between K values. For the final decision, we chose K =5 with a weighted version. After that this model version tested on our test data and their results are given below.



Prediction \ Ground Truth	Galaxy	QSO	Stars
Galaxy	4678	146	416
QSO	47	316	0
Stars	133	16	879

Classes	precision	recall	F1 score
Galaxy	0.962948	0.892748	0.92652
QSO	0.661088	0.870523	0.751486
Stars	0.678764	0.855058	0.75678

Figure 10: Confusion Matrix, Precision, Recall and F1 Scores of the KNN Model where K=5 and Weighted Version

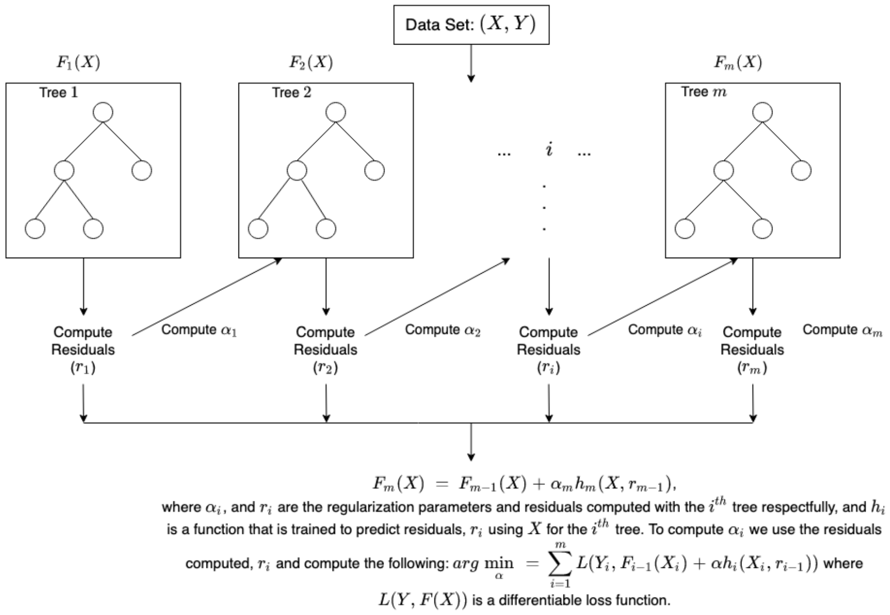
Test Accuracy = 0.89

For KNN 0.89 accuracy seems promising. As seen from the confusion matrix Galaxy is in very good shape but the rest have some problems. This can be observed from the scores table as well. Scores of Galaxy are very high which is good, however, QSO and Stars are relatively low. The model mostly labels Stars and QSO as Galaxy. It is because Galaxy is dominant in the dataset %71 of it. Since the model looks for nearest points, the galaxy can dominate those nearest neighbors as well which can cause this issue.

4.3. 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.

We have chosen the XGBoost because it is one of the trending classification algorithms based on decision trees. It wins Kaggle competitions so we considered that it can be used for our data as well.

Current XGBoost library includes hardware optimizations and additional sophisticated algorithms for split search and regularization. However, we have implemented the first version described in the paper “XGBoost: A Scalable Tree Boosting System”[9]. Our mathematics depends on this paper and formulas can be found there. Since we do not have hardware optimizations and our code was directly written in Python, our XGBoost algorithm is very slow. One model to train on training data takes more than 8h. This value also depends on the parameters. This mentioned 8h derived from our minimal parameters like depth = 3 and number of trees = 10. Therefore, we could not apply the k-fold algorithm to it. We tried 3 different XGBoost model parameter combinations to at least tune to some extent. We do not have enough computational power and time to tune further.



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

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 11: XGBoost Pseudo Algorithm [8]

4.4.1 Results

XGBoost has 7 parameters which are number of trees, maximum depth of one tree, minimum leaf size, learning rate, gamma, lambda, and columns subsample amount. Our gamma and lambda values are regularization parameters mentioned in the paper. To further add differences between trees we subsampled the columns. col_num parameter represents the number of columns that should be subsampled from features of input data. We have tried with half of the training data to make training time reasonable. Below are tried parameter combinations.

1-*N_trees = 20, max_depth = 10, min_leaf_size = 400, lr = 0.1, gamma = 1, lmb = 1, col_num = 4*
2-*N_trees = 20, max_depth = 10, min_leaf_size = 400, lr = 0.01, gamma = 5, lmb = 1, col_num = 4,*
3- *N_trees = 10, max_depth = 3, min_leaf_size = 400, lr = 0.1, gamma = 1, lmb = 1, col_num = 4,*

We compared these 3 models with validation test accuracy results as follows:

Validation accuracy of 1 : %94
Validation accuracy of 1 : %96
Validation accuracy of 1 : %97

Therefore the third parameter combination was chosen. Then we trained XGBoost with the 3rd set of parameters and evaluated it on the test set. Both the train and test sets are the same sets used in the Neural Network and k-NN models.

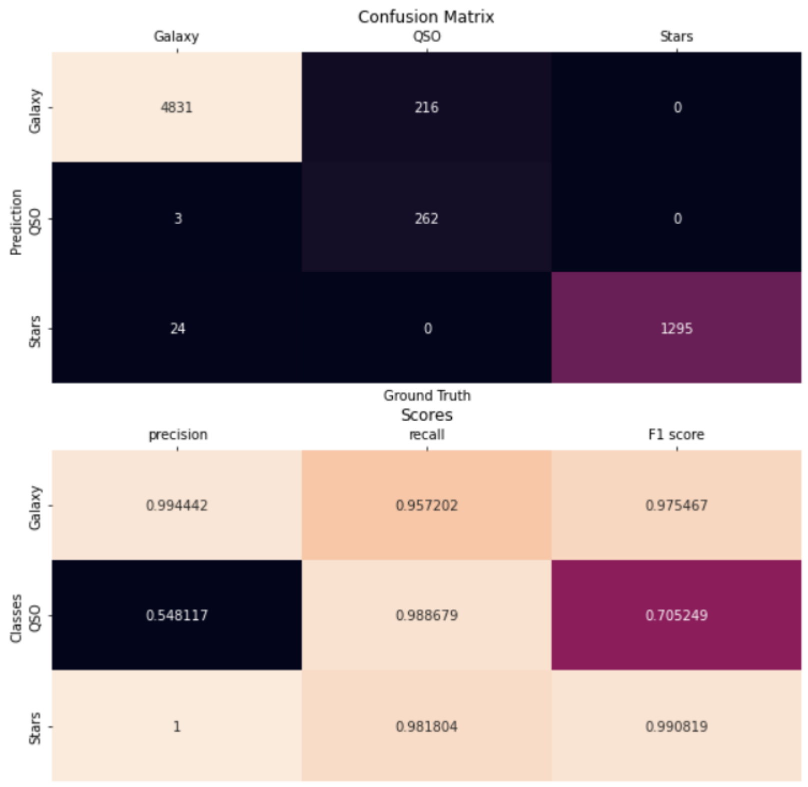


Figure 12: Confusion Matrix, Precision, Recall and F1 Scores of the Best XGBoost Model

As seen from figure 12, the model reaches 96% accuracy rate. It also predicts Galaxy and Star classes with a very high confidence. Indeed, it never misses Star class in the test set. However, it cannot properly predict the QSO class since it is a rare class. Furthermore, as explained above, since the XGBoost is trained on half of the data, it cannot predict the QSO class.

Training times of the models given as follows:

Model 1 → 1h 28mins
Model 2 → 1h 5mins
Model 3 → 1h 31mins

5. Gantt Chart

Work Packages	Task Owner(s)	Nov 1-5	Nov 6-11	Nov 12-17	Nov 17-20	Nov 21-26	Nov 22-30	Dec 1-6	Dec 7-12	Dec 13-19
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	Fuat, Melih			🟨	🟨	🟨	🟨	🟨	🟨	🟨
Code Debugging	Fuat, Melih						🟥	🟥	🟥	🟥
Evaluator	Melih		🟦							

Figure 11: Gantt Chart for The Project Progress

6. Faced Challenges and Conclusion

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 implemented very powerful and robust models except k-NN.

Our results are very promising which reveals that we did something correctly. Even though there is a high imbalance in the dataset, we tackled this problem excellently using Neural Network. All the models were optimized using Grid Search with k Fold CV except XGBoost. Grid Search is a very costly optimization technique, however, we managed to implement it very effectively.

XGBoost was a very complicated and sophisticated model to implement. Therefore, we consumed lots of time to write it from scratch and debug its code. Also, we couldn't write the code of XGBoost very efficiently, thus, it took so much time to train.

Our final scores are shown in the table below. The table shows that the Neural Network algorithm is better in terms of not only accuracy but also recall, precision and F1 score. It is not surprising since NNs are very robust and powerful models. k-NN has the lowest accuracy, precision, recall and F1 score compared to XGBoost and NN as expected since there is no learning period. It is also a very simple algorithm and uses only 2 parameters. Therefore, it cannot be optimized further as other complicated models. XGBoost yielded very satisfying results even though it is not optimized using Grid Search and it didn't use all the training data. It almost reaches the same accuracy levels and close precision, recall and F1 scores as in NN. Indeed, it prevails over the Neural Networks on Average Recall score. However, it cannot predict the lowest class QSO due to the lack of data. But, it is very confident about Galaxy and Star. So, it could have reached much better results than the Neural Network algorithm if it was trained with the whole dataset and optimized via k Fold CV.

Models	**Accuracy**	**Average Precision**	**Average Recall**	**Average F1**
Neural Network	**96%**	**0.929355**	**0.91595**	**0.9224697**
XGBoost Classifier	96%	0.8475197	0.975895	0.8905117
k-Nearest Neighbor	89%	0.76760002	0.87277663	0.81159549

Table 1: Accuracy, Average Precision, Recall and F1 Scores of All Models

References

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

[2] 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].

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

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

[5]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].

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

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

[8]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].

[9] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

CODE

All the codes are also 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,argmax_flag = False):
        self.argmax_flag = argmax_flag



    def eval(self,model,X, y, cost_metric, num_folds = 3,*argv):
        """
        real function to run CV alorithm
        """
        acc_list = []
        cost_print = True
        if cost_metric == 'CrossEntropy':
            cost = Cross_Entropy_Loss
        elif cost_metric == 'MSE':
            #mse = ((pred - y_val)**2).mean(axis=0)
            pass
        else:
            cost_print = False

        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)
            if self.argmax_flag:

                loss = cost(pred, y_val.T)
                arg_pred = np.argmax(pred,axis=0)
                true_label = np.argmax(y_val,axis = 1)
                total += loss/pred.shape[1]

            else:

                arg_pred = pred
                true_label = y_val.flatten()

            acc = np.sum(arg_pred == true_label)/arg_pred.shape[0]
            acc_list.append(acc)
            #print("Prediction shape: ", arg_pred.shape)


        if cost_print:
            print("Average loss:", np.round(total/num_folds,4) )
        print(f"Accuracy of folds respectively :"
,np.round(acc_list,2)*100)
        #print('\n')
        return total/num_folds, np.array(acc_list)


#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
```

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

    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)
```

Feature Engineering

```
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

    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)
```

KNN

```
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 np.array(predictions)
```

Neural Network

Layers

```
"""
This script includes neural network layers. For now there is just Dense
layer.
"""
import numpy as np
from utils import softmax, stable_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 == "stable_softmax":
            A = stable_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(self.Z, dA)

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

Trainer

```
"""
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 stable_softmax(x):
    eps = 1e-15
    e_x = np.exp(x - np.max(x))
    return e_x / (e_x.sum(axis=0) + eps)

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
```

XGBoost

```
from tqdm import tqdm
import numpy as np

class Node:
    def __init__(self, X, y, gradient, hessian, max_depth, min_leaf_size,
gamma, lmb, cover, num_class, col_num):
        self.X = X
        self.y = y
        self.max_depth = max_depth
        self.min_leaf_size = min_leaf_size
        self.left = None
        self.right = None
        self.split_feature = None
        self.split_value = None
        self.gain = None
        self.prediction = None
        self.depth = None
        self.gamma = gamma
        self.gradient = gradient
        self.hessian = hessian
        self.lmb = lmb
        self.is_leaf = False
        self.weight = None
        self.cover = cover
        self.num_class = num_class
        self.col_num = col_num
        self.column_subsample = np.random.permutation(X.shape[1])[:col_num]

    @staticmethod
    def loss_reduction(lhs_gradient, lhs_hessian, rhs_gradient,
rhs_hessian, lmb, gamma):
        left = (lhs_gradient ** 2) / (lhs_hessian + lmb)
        right = (rhs_gradient ** 2) / (rhs_hessian + lmb)
        tot = (lhs_gradient + rhs_gradient) ** 2 / (lhs_hessian +
rhs_hessian + lmb)
        gain =  1 / 2 * (left + right - tot) - gamma
        return gain

    def find_best_split(self):

        gain = np.zeros(self.num_class)
        # gain = gain * float("-inf")

        for col in self.column_subsample:
            #sorted_row = self.X[np.argsort(self.X[:, col])]
            sorted_row = np.unique(self.X[:,col])
            #for row in range(self.X.shape[0]):
            for value in sorted_row:
                # lhs = self.X[:, col] < sorted_row[row, col]
                # rhs = self.X[:, col] >= sorted_row[row, col]
                lhs = self.X[:, col] < value
                rhs = self.X[:, col] >= value

                if lhs.sum() < self.min_leaf_size or rhs.sum() <
self.min_leaf_size:
                    continue
                lhs_gradient = self.gradient[lhs].sum(axis=0)
                lhs_hessian = self.hessian[lhs].sum(axis=0)
                rhs_gradient = self.gradient[rhs].sum(axis=0)
                rhs_hessian = self.hessian[rhs].sum(axis=0)
                temp_gain = self.loss_reduction(lhs_gradient, lhs_hessian,
rhs_gradient,
                                                rhs_hessian,
self.lmb,self.gamma)

                if temp_gain.sum() > gain.sum():
                    #print("içerdeyim baba")
                    gain = temp_gain
                    self.split_feature = col
                    #self.split_value = self.X[row, col]
                    self.split_value = value

        if gain.all()  == np.zeros(self.num_class).all():
            self.is_leaf = True
            #self.calculate_weight()
        self.gain = gain

    def split(self):


        self.is_leaf_node()

        if not self.is_leaf:
            lhs = self.X[:, self.split_feature] < self.split_value
            rhs = self.X[:, self.split_feature] >= self.split_value

            self.left = Node(self.X[lhs], self.y[lhs], self.gradient[lhs],
self.hessian[lhs],
                             self.max_depth - 1,
self.min_leaf_size,self.gamma,
                             self.lmb, self.cover, self.num_class,
self.col_num)

            self.right = Node(self.X[rhs], self.y[rhs], self.gradient[rhs],
self.hessian[rhs],
                              self.max_depth - 1, self.min_leaf_size,
self.gamma,
                              self.lmb, self.cover, self.num_class,
self.col_num)
        else:
            self.calculate_weight()

    def is_leaf_node(self):

        #or self.weight < self.cover
        if self.max_depth <= 0 :
            #self.calculate_weight()
            self.is_leaf == True

    def calculate_weight(self):


        gradient = np.sum(self.y * self.gradient, axis=0)
        hessian = np.sum(self.hessian, axis=0)
        self.weight =  -gradient / (hessian + self.lmb)

        #self.weight = - self.gradient.sum(axis=0) /
(self.hessian.sum(axis=0) + self.lmb)


    def predict_sample(self, sample):
        if self.is_leaf:
            return self.weight

        if sample[self.split_feature] <= self.split_value:
            node = self.left
        else:
            node = self.right
        return node.predict_sample(sample)

    def predict(self, X):

        prediction = np.zeros((X.shape[0], self.num_class))
        for k, sample in enumerate(X):
          #  print(sample)
            prediction[k] = self.predict_sample(sample)
       # print(prediction[-5:-1])
        return prediction

class Tree:
    def __init__(self, X, y, gradient, hessian, max_depth, min_leaf_size,
gamma, lmb, cover, num_class, col_num):
        self.root = Node(X, y, gradient, hessian, max_depth, min_leaf_size,
gamma, lmb, cover, num_class, col_num)
        #self.root.calculate_weight()
        self.root.depth = 0
        self.max_depth = max_depth
        self.min_leaf_size = min_leaf_size
        self.gamma = gamma
        self.lmb = lmb
        self.cover = cover
        self.num_class = num_class
        self.col_num = col_num

    def grow_tree(self, node):


        node.find_best_split()
        node.split()

        if node.is_leaf:
            return

        self.grow_tree(node.left)
        self.grow_tree(node.right)


    def predict(self, X):
        return self.root.predict(X)

    def learn(self):


        self.grow_tree(self.root)

class XGBoost_Classifier:
    def __init__(self, n_trees=5, max_depth=3, min_leaf_size=400,
                 learning_rate=0.3, gamma=0, lmb=1, cover=1, col_num=4):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.min_leaf_size = min_leaf_size
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.lmb = lmb
        self.cover = cover
        self.col_num = col_num

    def gradient_hessian(self, y, y_pred):


        gradient = y_pred - y
        hessian = y_pred * (1 - y_pred)
        return gradient, hessian

    def learn(self, X, y):

        self.trees = []
        self.X = X
        self.y = y
        self.n_classes = len(np.unique(y, axis=0))
        #self.init_prediction = np.array([1/self.n_classes] *
(self.y.shape[0]*self.y.shape[1])).reshape(self.y.shape)
        self.init_prediction = np.random.rand(self.y.shape[0],
self.n_classes)
        self.init_prediction = self.stable_softmax(self.init_prediction)

        self.y_pred = self.init_prediction
        for i in tqdm(range(self.n_trees)):
            gradient, hessian = self.gradient_hessian(self.y, self.y_pred)

            tree = Tree(self.X, self.y, gradient, hessian, self.max_depth,
                        self.min_leaf_size, self.gamma, self.lmb,
                        self.cover, self.n_classes, self.col_num)
            tree.learn()
            self.trees.append(tree)
            self.y_pred += self.learning_rate * tree.predict(self.X)
            self.y_pred = self.stable_softmax(self.y_pred)


    @staticmethod
    def stable_softmax(x):

        e_x = np.exp(x - np.max(x, axis = 1, keepdims = True))
        return e_x / e_x.sum(axis = 1, keepdims = True)

    def predict(self, X,argmax=True):

        y_pred1 = np.zeros((X.shape[0], self.n_classes))
        for tree in self.trees:
            y_pred1 += self.learning_rate * tree.predict(X)

        y_pred = self.stable_softmax(y_pred1)
        if argmax:
            return np.argmax(y_pred, axis=1)
        return y_pred1
```

# Implementation Notebook code, It call above classes to run all the methods explained in the report

```
!git clone https://github.com/fuat-arslan/ML.git

#!pip install
https://github.com/pandas-profiling/pandas-profiling/archive/master.zip

# Commented out IPython magic to ensure Python compatibility.
import numpy as np
import pandas as pd
import time
from pandas_profiling import ProfileReport
from matplotlib import pyplot as plt

# %matplotlib inline

data_raw = pd.read_csv("/content/ML/raw_data/archive/sdss-IV-dr16-70k.csv")
```

```python
data_raw.head()

profile = ProfileReport(data_raw)
profile

"""# PREPROCESSING"""

# Commented out IPython magic to ensure Python compatibility.
# %cd ML

from Preprocess.Preprocess import pd_np_Converter, OutlierRemoval, MinMax,
Encoder, StratifiedTrainValTestSplit
from Preprocess.Feauture_Engineering import PCA

# rerun is constant, objid is not a feature
data = data_raw.drop(columns=["rerun", "objid"])
# move class to end of data
label = data.pop("class")
data["class"] = label
data.head()

# convert data to numpy
converter = pd_np_Converter(data)
data_np = converter.to_nup()
X = data_np[:, :-1].astype(float)
Y = data_np[:, -1].reshape(-1,1)
X, Y

# then apply outlier removal and standard normalized
OutRemove = OutlierRemoval(X, Y, 3) # 3 denotes threshold value
X_clean, Y_clean = OutRemove.fast()
X_clean.shape, Y_clean.shape

# since psf_Mag's are highly linearly correlated, they are decorrelated
using PCA
# we are not decreasing the dimension!!
pca = PCA()
decorrelated_psf = pca.fast(X_clean[:, 2:7], 5)
decorrelated_psf
# decorrelated values are very small, since all other features are between
-3,3;
# decorrelated values are linearly mapped to -3,3 range back
# map decorrleted_psf to -3,3
mapper = MinMax(low = -3, high = 3)
psf = mapper.fast(decorrelated_psf)
psf

# change old values by new decorrelated values
X_clean[:, 2:7] = psf

# label encoder
encode1 = Encoder(one_hot = False)
Y = encode1.fast(Y_clean)
Y.shape

"""# Train Validation Test Split"""

splitter = StratifiedTrainValTestSplit(X_clean, Y, train_size=0.8,
                                       val_size=0.1, test_size=0.1,
                                       stratify=True, random_state=1)

X_train, X_val, X_test, Y_train, Y_val, Y_test = splitter.split()
X_train.shape, X_val.shape, X_test.shape, Y_train.shape, Y_val.shape,
Y_test.shape

"""# k-NN"""

from Models.KNN import KNN
from Metrics.Metrics import kFold, Evaluator

X_cv = np.vstack((X_train, X_val))
Y_cv = np.vstack((Y_train, Y_val))
X_cv.shape, Y_cv.shape

# argmax_flag denotes labels are categoric, not one_hot
k_list = [5,7,9,11,13,15,17,19,21]
acc_dict = {}
for k in k_list:
    t1 = time.time()
    print("Cross validation is being done for k="+str(k))
    cv_k_NN = kFold(argmax_flag = False)
    k_NN_model = KNN(num_neig = k, metric = "euclidian", weighted = True)
    avg_loss, acc_list = cv_k_NN.eval(k_NN_model, X_cv, Y_cv, None, 5)
    acc_dict[str("k = ")+str(k)] = acc_list
    t2 = time.time()
    print(f"Elapsed time for CV when k = {k} is {np.round(t2-t1,2)}
secs\n")

avg_dict = {key: np.sum(value) / len(value) for key, value in
acc_dict.items()}
avg_dict

avg_acc = 100*np.round(list(avg_dict.values()),4)
plt.plot(k_list, avg_acc, "x")
plt.xlabel("k")
plt.ylabel("Accuracy (%)")
plt.title("Accuracy vs k Plot")

# for unweighted k-NN
k_list = [5,7,9,11,13,15,17,19,21]
acc_dict = {}
for k in k_list:
    t1 = time.time()
    print("Cross validation is being done for k="+str(k))
    cv_k_NN = kFold(argmax_flag = False)
    k_NN_model = KNN(num_neig = k, metric = "euclidian", weighted = False)
    avg_loss, acc_list = cv_k_NN.eval(k_NN_model, X_cv, Y_cv, None, 5)
    acc_dict[str("k = ")+str(k)] = acc_list
    t2 = time.time()
    print(f"Elapsed time for CV when k = {k} is {np.round(t2-t1,2)}
secs\n")

"""**Testin k-NN on Validated Parameters**"""

# we found that the most optimal k = 5
optimal_k_NN = KNN(num_neig = 5, metric = "euclidian", weighted = True)
pred_k_NN = optimal_k_NN.learn(X_train, Y_train).predict(X_test)
eval = Evaluator(pred_k_NN, Y_test, label_dict = {"Galaxy": 0, "QSO": 1,
"Stars": 2})

_,_ = eval.scores()

"""# Neural Network"""

from Models.NeuralNetwork.Layers import Dense
from Models.NeuralNetwork.Optimizer import Optimizer
from Models.NeuralNetwork.Trainer import NeuralNetwork
```

```
# label encoder for one hot
encoder = Encoder(one_hot = True)
Y2 = encoder.fast(Y_clean)
Y2.shape

splitter = StratifiedTrainValTestSplit(X_clean, Y2, train_size=0.8,
                                       val_size=0.1, test_size=0.1,
                                       stratify=True, random_state=1)

X_train, X_val, X_test, Y_train, Y_val, Y_test = splitter.split()
X_train.shape, X_val.shape, X_test.shape, Y_train.shape, Y_val.shape,
Y_test.shape

X_cv = np.vstack((X_train, X_val))
Y_cv = np.vstack((Y_train, Y_val))
X_cv.shape, Y_cv.shape

# define grid search on the parameters, there are 64 possibilities
param_grid = {}
param_grid["learning_rates"] = [0.0001, 0.001, 0.01]
param_grid["batch_sizes"] = [16, 32, 64]
param_grid["layer_neuron"] = [ [14, 32, 8, 3], [14, 16, 32, 3], [14, 64,
32, 8, 3] ]
param_grid["activations"] = ["relu", "sigmoid", "tanh"]

param_combinations =
np.array(np.meshgrid(*param_grid.values())).T.reshape(-1, len(param_grid))

scores = {}

print(f"There will be {param_combinations.shape[0]} combinations")
for j, params in enumerate(param_combinations):
        t1 = time.time()
        params_dict = {key: value for key, value in zip(param_grid.keys(),
params)}

        # creating the NN model
        layers = []
        for i in range(1, len(params_dict["layer_neuron"])-1):
            layers.append(Dense(params_dict["layer_neuron"][i-1],
params_dict["layer_neuron"][i], activation = params_dict["activations"]))
        layers.append(Dense(params_dict["layer_neuron"][-2],
params_dict["layer_neuron"][-1], activation = "stable_softmax"))
```

```
        model_NN = NeuralNetwork(layers, "CrossEntopy")
        adam = Optimizer("adam", params_dict["learning_rates"], None, 0.9,
0.99)

        # creating kFold
        cv_NN = kFold(True)
        avg_loss, acc_list = cv_NN.eval(model_NN, X_cv, Y_cv,
                                        "CrossEntropy",
                                        5,
                                        adam,
                                        21,
                                        params_dict["batch_sizes"])

        # Store the mean score for this combination of hyperparameters
        scores[str(j)] = np.mean(acc_list)

        t2 = time.time()
        print(f"Elapsed time for {j}th parameter set is {np.round(t2-t1,2)}
secs")

# Find the index of the best combination of hyperparameters
print(scores)
max_idx = max(scores, key=scores.get)
params = param_combinations[int(max_idx)]
optimal_dict = {key: value for key, value in zip(param_grid.keys(),
params)}
print(optimal_dict)

scores = {'0': 0.9705813369073546, '1': 0.9651867984587031, '2':
0.9644664097838834, '3': 0.9757413302060647, '4': 0.97543977215614, '5':
0.9746356173563411, '6': 0.7325347629418664, '7': 0.9754565253811359, '8':
0.9739989948065002, '9': 0.9690902998827274, '10': 0.9680515999329872,
'11': 0.9582007036354498, '12': 0.976059641480985, '13': 0.976076394705981,
'14': 0.9753057463561736, '15': 0.7325347629418664, '16':
0.9761266543809682, '17': 0.9746021109063495, '18': 0.9740492544814877,
'19': 0.9727257497068186, '20': 0.9700619869324845, '21':
0.975707823756073, '22': 0.9750376947562407, '23': 0.9746858770313285,
'24': 0.7325347629418664, '25': 0.9744513318813871, '26':
0.9739989948065002, '27': 0.964583682358854, '28': 0.9303903501424025,
'29': 0.8823085944044229, '30': 0.976428212430893, '31': 0.976327693080918,
'32': 0.975087954431228, '33': 0.9754732786061318, '34':
0.9763109398559223, '35': 0.9754230189311442, '36': 0.9629418663092647,
'37': 0.9269391857932652, '38': 0.8902998827274251, '39':
0.9760261350309936, '40': 0.975004188306249, '41': 0.9749204221812701,
'42': 0.9740660077064834, '43': 0.9701289998324677, '44':
0.9749874350812531, '45': 0.9710336739822416, '46': 0.9686547160328363,
'47': 0.9029318143742671, '48': 0.9751047076562237, '49':
0.9756073044060981, '50': 0.9753224995811693, '51': 0.9749706818562573,
'52': 0.9765957446808511, '53': 0.9748366560562909, '54':
0.9722734126319317, '55': 0.9692745853576813, '56': 0.9685541966828615,
'57': 0.9773663930306583, '58': 0.9758250963310438, '59':
0.9758418495560395, '60': 0.9728262690567935, '61': 0.9744680851063829,
'62': 0.9752387334561903, '63': 0.9721393868319652, '64':
0.968956274082761, '65': 0.9641648517339588, '66': 0.9759591221310103,
'67': 0.9753392528061651, '68': 0.9750209415312447, '69':
0.9740157480314962, '70': 0.9704305578823924, '71': 0.9732786061316805,
'72': 0.9755067850561232, '73': 0.9737309432065674, '74':
0.972424191656893, '75': 0.9757413302060647, '76': 0.9753224995811693,
'77': 0.9715697771821075, '78': 0.9689730273077568, '79':
0.9700284804824928, '80': 0.972507957781873}

# Sort the scores dictionary by value to see how much better the best model
sorted_scores = {k: v for k, v in sorted(scores.items(), key=lambda item:
item[1], reverse = True)}
sorted_scores

"""**Testing Neural Network on Validated Parameters**"""

layers = [Dense(14, 32, activation = "tanh"),
          Dense(32, 8, activation = "tanh"),
          Dense(8, 3, activation = "softmax")]

model_NN = NeuralNetwork(layers, "CrossEntopy")
adam = Optimizer("adam", 0.001, None, 0.9, 0.99)

model_NN.learn(X_train, Y_train, optimizer = adam, max_epoch = 90,
batch_size = 16)

pred = model_NN.predict(X_test)

evaluate = Evaluator(np.argmax(pred.T, axis = 1), np.argmax(Y_test, axis =
1),
                     label_dict = {"Galaxy": 0, "QSO": 1, "Stars": 2})
_,_ = evaluate.scores()
```

```
"""# XGBoost """

from Models.XGBoost import XGB
xgb = XGB.XGBoost_Classifier(n_trees = 10, max_depth = 3, min_leaf_size =
400,
                             learning_rate = 0.1, gamma = 1, lmb=1, col_num
= 4)

t1 = time.time()
xgb.learn(X_train, Y_train)
t2 = time.time()
print(t2-t1)

pred_train = xgb.predict(X_train)
evaluate_train = Evaluator(pred_train, np.argmax(Y_train, axis = 1),
                     label_dict = {"Galaxy": 0, "QSO": 1, "Stars": 2})
_,_ = evaluate_train.scores()

pred_val = xgb.predict(X_test)
evaluate_val = Evaluator(pred_val, np.argmax(Y_test, axis = 1),
                     label_dict = {"Galaxy": 0, "QSO": 1, "Stars": 2})
_,_ = evaluate_val.scores()

xgb2 = XGB.XGBoost_Classifier(n_trees = 15, max_depth = 5, min_leaf_size =
200,
                             learning_rate = 0.3, gamma = 1, lmb=1, col_num
= 4)

t1 = time.time()
xgb2.learn(X_train, Y_train)
t2 = time.time()
print(t2-t1)

pred_train = xgb2.predict(X_train)
evaluate_train = Evaluator(pred_train, np.argmax(Y_train, axis = 1),
                     label_dict = {"Galaxy": 0, "QSO": 1, "Stars": 2})
_,_ = evaluate_train.scores()

pred_val = xgb2.predict(X_test)
evaluate_val = Evaluator(pred_val, np.argmax(Y_test, axis = 1),
                     label_dict = {"Galaxy": 0, "QSO": 1, "Stars": 2})
_,_ = evaluate_val.scores()
```

```
xgb3 = XGB.XGBoost_Classifier(n_trees = 20, max_depth = 10, min_leaf_size =
400,
                             learning_rate = 0.01, gamma = 5, lmb=1,
col_num = 4)

t1 = time.time()
xgb3.learn(X_train_small, Y_train_small)
t2 = time.time()
print(t2-t1)

pred_val3 = xgb3.predict(X_val)
evaluate_train = Evaluator(pred_val3, np.argmax(Y_val, axis = 1),
                     labeldict = {"Galaxy": 0, "QSO": 1, "Stars": 2})
_28,_ = evaluate_train.scores()
```