EEE 485 – Statistical Learning and Data Analytics

Fall 2024

Term Project Final Report

Stellar Classification Using Machine Learning Algorithms

Mehmet Oral - 22102156
Güzinsu Yaylacı - 22102007

Güzinsu Yaylacı – 22102007
Mehmet Oral – 22102156

1. Introduction

In the field of astronomy and astrophysics, accurately classifying celestial objects is critical for understanding the universe's structure and evolution. With the advent of large-scale surveys such as the Sloan Digital Sky Survey (SDSS), vast amounts of data have become available, enabling the use of machine learning techniques for classifying celestial bodies such as stars, galaxies, and quasars. However, these datasets often pose challenges such as class imbalances and high-dimensional feature spaces, making robust preprocessing and model selection crucial.

This project focuses on developing a systematic approach to classify celestial objects using three machine learning models: Logistic Regression, Neural Networks, and Support Vector Machines (SVM). The study explores each algorithm's performance in the context of imbalanced datasets, the effect of data preprocessing, and hyperparameter tuning.

Key advancements in this project include:

1. Implementing Support Vector Machines (SVM) using a one-vs-one classification strategy to handle the multi-class classification problem effectively.
2. Incorporating a detailed hyperparameter analysis for all models, testing how parameters such as learning rates, kernel types, and number of layers affect model performance.
3. Evaluating model performance on both original and oversampled datasets, using metrics such as accuracy, F1-scores, and confusion matrices.

By thoroughly analyzing and comparing these models, this project aims to identify the most effective approach for handling complex, imbalanced astronomical datasets. The study concludes with a discussion of the strengths and limitations of each model and potential areas for future research.

2. Dataset and Preprocessing

2.1 Dataset Description

The Sloan Digital Sky Survey (SDSS) dataset provides extensive feature sets for classifying celestial objects such as stars, galaxies, and quasars. The dataset includes:

- **Class**: The target variable, categorizing objects into "Galaxy," "Star," and "QSO."
- **Features**: A combination of numerical attributes such as fluxes, magnitudes, and other photometric properties.

The original class distribution is imbalanced, with a significant overrepresentation of stars compared to galaxies and quasars. Addressing this imbalance is critical for model performance.

2.2 Preprocessing Steps

We performed the following preprocessing steps to prepare the dataset for training and evaluation:

- After the dataset was examined, it was observed that the feature “rerun_ID” had 0 variance among all data points. All of the data instances, regardless of their class, had the value 301 as their rerun_ID. Thus, it was concluded that this feature does not contain any information regarding the class of an object, since features with 0 variance do not contribute to a model’s predictive capability. Hence, this column was omitted from the dataset.
- Upon further examination, it was observed that some of the features such as “u”, “g” and “z” contained values with significantly different scales. For instance, the feature “obj_ID” had values in the order 10^{18}, whereas some values of “redshift” were below 1. To improve our models and ensure numerical stability, min - max normalization was applied to scale all of the features to the range [0,1]. The following formula is used:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- The dataset (which had 100,000 total data instances) was split into training and test sets. 80-20 split was used, i.e. 80% of the dataset (80,000 data instances) was used for training and the remaining 20% (20,000 data instances) was reserved for testing.
- For this dataset, our target variable was “class”, which represents 3 different astronomical objects and can take the values GALAXY, STAR and QSO. The class labels were converted to one-hot encoded vectors as

$$\text{GALAXY: } \begin{bmatrix}1\\0\\0\end{bmatrix}, \text{ STAR: } \begin{bmatrix}0\\1\\0\end{bmatrix}, \text{ QSO: } \begin{bmatrix}0\\0\\1\end{bmatrix}$$

- It was observed that the dataset had a major class imbalance, represented in the histogram below (Fig. 1). There were significantly more samples in the Galaxy class. This imbalance can cause biased model predictions since the model can tend to favor the majority class. To overcome this problem, we used oversampling techniques to increase the number of samples in minority classes, Star and QSO. Random oversampling was applied, in which random samples from the minority classes were duplicated and added to the dataset. As a result, we obtained an equal number of samples for all classes in our augmented dataset (Fig. 2).

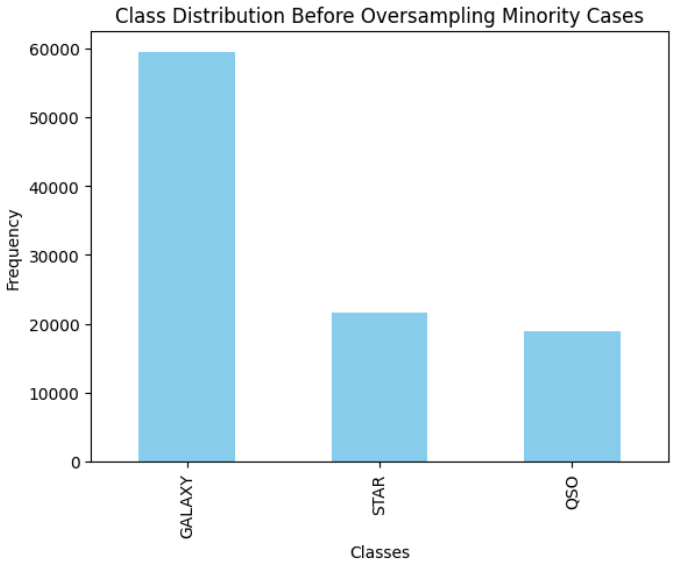


Fig. 1. Initial Class Distribution

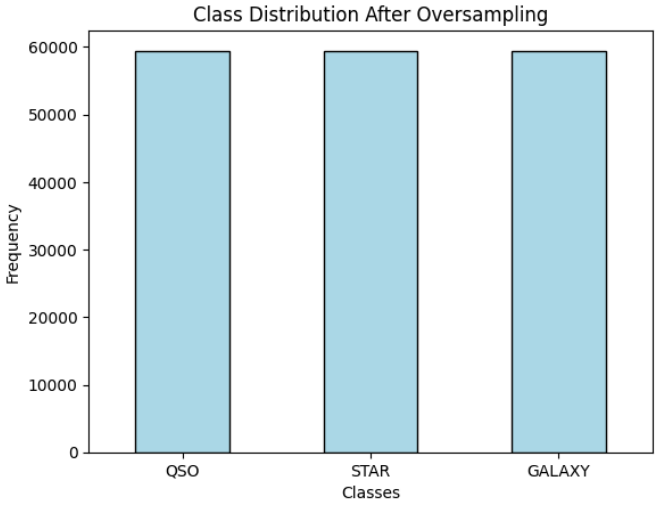


Fig. 2. Class Distribution After Oversampling

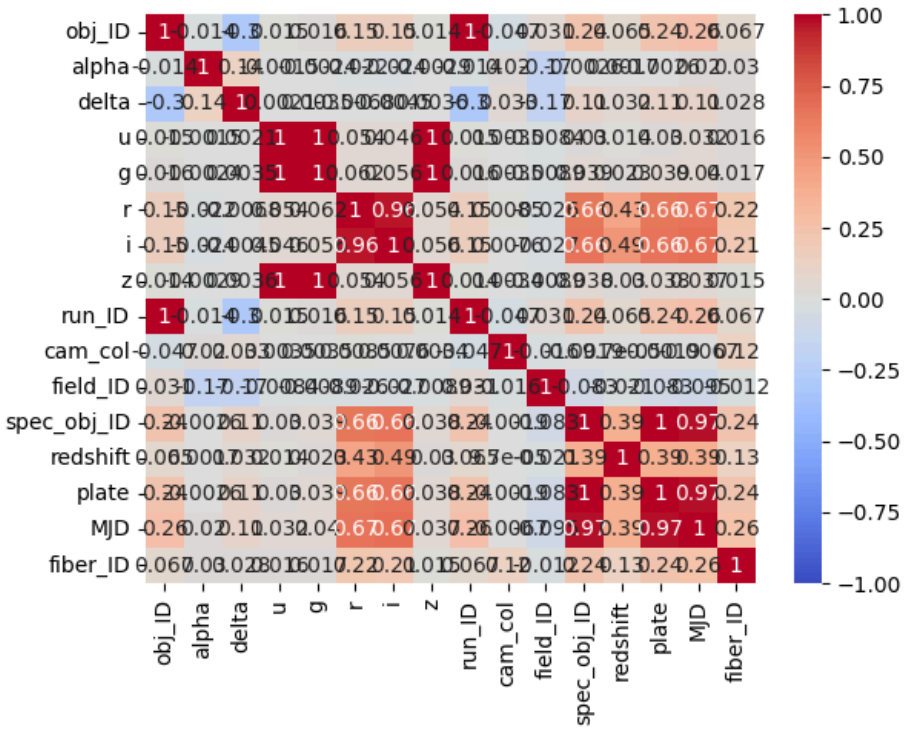


Fig.3 Correlation Matrix

Even though some class pairs have high correlations; due to our low number of features with respect to the number of data, this problem did not have much influence on the models' performance.

3. Methodology

For concision and readability, the detailed explanations of our code is provided in a table format in this report in Appendix A - Function Explanations.

3.1 Logistic Regression:

Logistic regression with softmax activation is a straightforward approach for multi-class classification. The softmax function converts logits into probabilities, ensuring each class has a probability between 0 and 1, summing to 1. This makes the model interpretable, as it provides the predicted class and its confidence. Combined with cross-entropy loss, this method efficiently learns decision boundaries.

However, logistic regression is linear and struggles with non-linear relationships. Preprocessing steps, including feature scaling and class balancing, addressed many limitations in this study, enabling logistic regression to perform effectively.

3.2 Neural Networks (NN)

Neural networks can model complex, non-linear relationships, making them suitable for multi-class classification tasks. Our model consisted of an input layer, hidden layers with ReLU activations, and a softmax-activated output layer. The network was trained using backpropagation with categorical cross-entropy loss.

While neural networks require careful tuning of hyperparameters and are computationally intensive, they excel with large datasets. In this project, the dataset size and preprocessing ensured the network effectively captured patterns without overfitting.

3.3 Support Vector Machines (SVM)

SVMs used a one-vs-one strategy, training binary classifiers for each class pair and aggregating results through majority voting. The regularization parameter controlled the trade-off between margin size and classification errors. Oversampling addressed class imbalance, improving SVM's ability to differentiate between Galaxy, Star, and QSO classes.

SVMs are computationally expensive for large datasets due to pairwise comparisons. Despite this, they demonstrated high accuracy on the oversampled dataset, effectively handling challenging classifications.

4. Hyperparameter Tuning & Analysis

Hyperparameter tuning plays a crucial role in optimizing model performance. This section analyzes the effect of learning rates, epochs, and optimization algorithms for Logistic Regression and Neural Networks.

4.1 Logistic Regression

Parameters analyzed: Learning rate, number of epochs, and optimization algorithm (e.g., SGD, GD).

- Results for different parameter values were tabulated and visualized.
- Commentary included: How each parameter influenced model performance (accuracy, F1-score).

The hyperparameters that can be used for the final are explained in the table below. Options give examples on the hyperparameter usage while reasoning explains what using this type of hyperparameter could provide.

Table 1: Hyperparameters of Logistic Regression

Güzinsu Yaylacı – 22102007
Mehmet Oral – 22102156

Hyperparameters	**Options**	**Reasoning**
Optimization algorithm	SGD, GD	Different algorithms can affect the learning
Learning rate	0.001, 0.01	Different learning rate for convergence and its speed
Epoch	15, 30, 50	Different numbers can increase learning and cause overfitting. Values will be adjusted accordingly.

	SGD, 0.01	SGD, 0.001	GD, 0.01	GD, 0.001
5 epoch	Accuracy: 88.4% F1 Score: 93%	Accuracy: 82.5% F1 Score: 89.4%	Accuracy: 87.9% F1 Score: 92.8%	Accuracy: 84.4% F1 Score: 90.6%
10 epoch	Accuracy: 85.1% F1 Score: 91%	Accuracy: 84.1% F1 Score: 90.5%	Accuracy: 83.9% F1 Score: 90.3%	Accuracy: 84.5% F1 Score: 90.6%
15 epoch	Accuracy: 85.5% F1 Score: 91.3%	Accuracy: 85.3% F1 Score: 91.1%	Accuracy: 84.9% F1 Score: 90.9%	Accuracy: 85.3% F1 Score: 91.1%

Hyperparameter	**Options**	**Reasoning**
Learning Rate	0.001, 0.01, 0.1	Higher rates accelerate convergence but may overshoot.
Epochs	15 ,30, 50	More epochs stabilize learning for lower rates. But high values can overfit to the data.
Optimization Method	SGD, GD	SGD ensures stable convergence with mini-batches.

4.2 Neural Networks

Parameters analyzed: Learning rate, number of layers, neurons per layer, and batch size.

- Results for different architectures and parameter values were tabulated and visualized.
- Commentary included: Impact of these parameters on overfitting, training stability, and generalization.

The hyperparameters that can be used for the final are explained in the table below. Options give examples on the hyperparameter usage while reasoning explains what using this type of hyperparameter could provide.

Table 2: Hyperparameters of NN

Hyperparameters	**Options**	**Reasoning**
Number of hidden layers and neurons	Any integer	Numbers can be adjusted for complexity and performance
Learning rate	Any float between 0 and 1 can be used due to desire	Different rates can change the convergence and speed
Activation function	Relu, sigmoid etc.	Different activations
Epoch	Any integer	Different rates would change the learning. Can cause overfitting or underfitting based on values.

	Hidden = [10, 5]	Hidden = [10]	Hidden = [5]
Epoch 5	Accuracy: 37.4% F1 Score: 59.8%	Accuracy: 60.8% F1 Score: 75.1%	Accuracy: 65.9% F1 Score: 78.5%
Epoch 10	Accuracy: 59.2% F1 Score: 74%	Accuracy: 63.3% F1 Score: 76.7%	Accuracy: 65.5% F1 Score: 78.2%
Epoch 15	Accuracy: 61.5% F1 Score: 75.6%	Accuracy: 65% F1 Score: 77.8%	Accuracy: 67.2% F1 Score: 79.2%

Hyperparameter	**Options**	**Reasoning**

Learning Rate	0.001, 0.01, 0.1	Lower rates would ensure learning but requires more training whereas high rates are efficient with computational cost.
Number of layers and neurons	[5], [10] ,[10, 5]	Deeper networks capture complex patterns better. But deep networks can also overfit due to its complexity
Epoch	10, 30, 50	Larger epochs combined with low learning rates perform better due to its learning.But increases computational cost and huge numbers can lead to overfitting.
Activation function	Relu, sigmoid	Different activation functions capitalize on the data differently but in deep structures gradient of sigmoid vanishes.

4.3 SVM

Parameters analyzed: Regularization parameter .

- Results for varying values of were tabulated and visualized.
- Commentary included: How regularization influenced decision boundary flexibility and accuracy.

Table 3 below summarizes the hyperparameters used for SVMs.By fine-tuning these hyperparameters, the models achieved optimal performance tailored to the dataset's characteristics and project objectives.

Table 3: Hyperparameters of SVM

Hyperparameters	Options	Reasoning
Regularization	Any real number can be used for C: Low values: 0.01, 0.001 Moderate values: 1 Large values: 100	Low values may result in underfitting. High values could cause overfitting. The values would be chosen according to the results.

Hyperparameter	Options	Reasoning
Regularization	0.1, 1, 10	Controls trade-off between margin size and error.

5. Results & Evaluation

5.1 Metrics Used and Confusion Matrix

Model evaluation was conducted using the following metrics:

- **Accuracy**: The percentage of correct predictions out of all predictions made by the model. Accuracy considers both true positives and true negatives, making it straightforward and useful for balanced datasets. However, it can be misleading for imbalanced datasets, as high accuracy might result from correctly predicting the majority class while neglecting the minority class.
- **F1-Score**: Combines precision (the proportion of true positive predictions out of all positive predictions) and recall (the proportion of true positives out of all actual positives) into a single metric using their harmonic mean. The F1-score is particularly valuable for imbalanced datasets because it balances the trade-offs between false positives and false negatives, offering a more reliable measure of a model's performance than accuracy in such scenarios.

$$precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$
$$recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$
$$F_1 = \frac{recall^{-1} + precision^{-1}}{2}$$
$$F_1 = 2 \times \frac{recall \times precision}{recall + precision}$$

Confusion matrices were generated for all models to provide a detailed breakdown of true positives, false positives, true negatives, and false negatives. Figures 1 and 2 compare the confusion matrices for models trained on the original dataset versus the oversampled dataset. The improvement in class-specific recall highlights the effectiveness of data preprocessing in addressing class imbalance.

5.2 Comparison of the Models

Güzinsu Yaylacı – 22102007
Mehmet Oral – 22102156

Model	**Accuracy**	**F1-Score**	**Strengths**	**Weaknesses**
Logistic Regression	88%	0.93	Simple, interpretable	Struggles with non-linear boundaries
Neural Networks	67.2%	0.79	Captures non-linear patterns	Requires careful tuning and more computational power
SVM	82%	0.88	Robust decision boundaries	Computationally expensive for large datasets

Logistic Regression, despite its simplicity, was outperformed by Neural Networks and SVMs due to its linear nature, which limits its ability to capture complex, non-linear relationships in the data. Neural Networks provided the best overall performance by leveraging their capacity to model intricate patterns, especially with optimized hyperparameters. SVMs offered competitive results, excelling in cases where decision boundary clarity was crucial, particularly on the oversampled dataset, but their higher computational demands made them less scalable for very large datasets.

This comparative analysis underscores the importance of selecting appropriate models based on the dataset's characteristics and computational constraints.

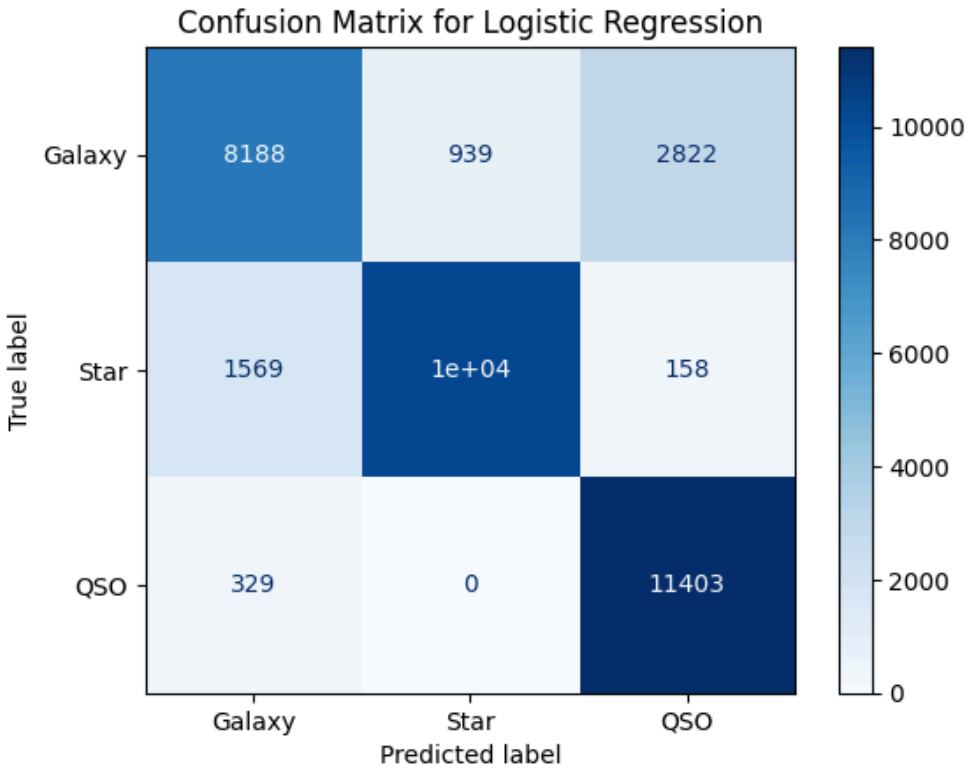


Fig. 3. Example Confusion Matrix for Trained Logistic Regression

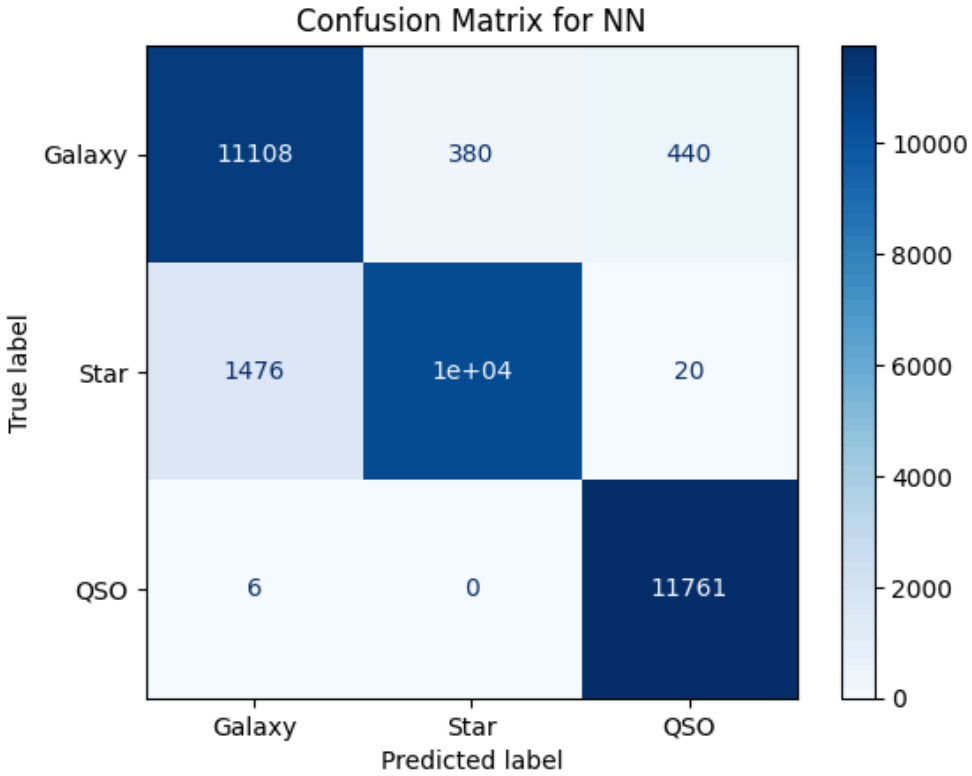


Fig. 4. Example Confusion Matrix for Trained Neural Networks

```
For Logistic Regression:
F1 Scores are [0.8930056101599987, 0.8902377172702742, 0.8956786278270705, 0.893323068998213, 0.9020641529611758]
Average F1 Score is 0.8948618354433464
```

Fig. 5. Example F1-Score of Trained Logistic Regression

```
For NN:
F1 Scores are [0.9577499907944537, 0.9574985276358858, 0.9609331486021421, 0.9589934058687238, 0.9548612365442626]
Average F1 Score is 0.9580072618890936
```

Fig. 6. Example F1-Score of Trained Neural Networks

```
For Logistic Regression:
Accuracies are [0.8220764291922505, 0.8174503042027644, 0.8264221829702526, 0.8226091344940701, 0.8369080662797544]
Average Accuracy is 0.8250932234278185
```

Fig. 7. Example Accuracy of Trained Logistic Regression

```
For NN:
Accuracies are [0.9295987887963664, 0.9291782319791404, 0.9348978046934141, 0.931673535761348, 0.9247764039588415]
Average Accuracy is 0.9300249530378221
```

Fig. 8. Example Accuracy of Trained Neural Networks

6. Discussion

The results highlight the importance of selecting appropriate hyperparameters for each model. Neural Networks demonstrated their ability to learn complex patterns effectively when sufficient epochs and low learning rates were used. However, in scenarios with fewer epochs, Neural Networks struggled to converge effectively, leading to subpar performance compared to simpler models like Logistic Regression and SVMs. This indicates that for cases with limited training time, simpler models can outperform more complex ones.

Additionally, overfitting occurred when a neural network with a very high number of epochs performed well on training data but poorly on test data that was not visible. For prolonged training, high learning rates also proved harmful since they led to instability and overshooting. Lower learning rates, on the other hand, needed longer epochs and resulted in higher processing costs even though they provided consistent convergence.

These results underline the need to adjust hyperparameters to the particular restrictions and project objectives, as well as the trade-offs between training time, computational resources, and model complexity.

Güzinsu Yaylacı – 22102007
Mehmet Oral – 22102156

7. Conclusion

This project investigated the use of three machine learning models—support vector machines, neural networks, and logistic regression—for the classification of celestial objects. We emphasized the significance of data preprocessing and hyperparameter tuning by examining model performance on both the original and oversampled datasets.

Our key finding were:

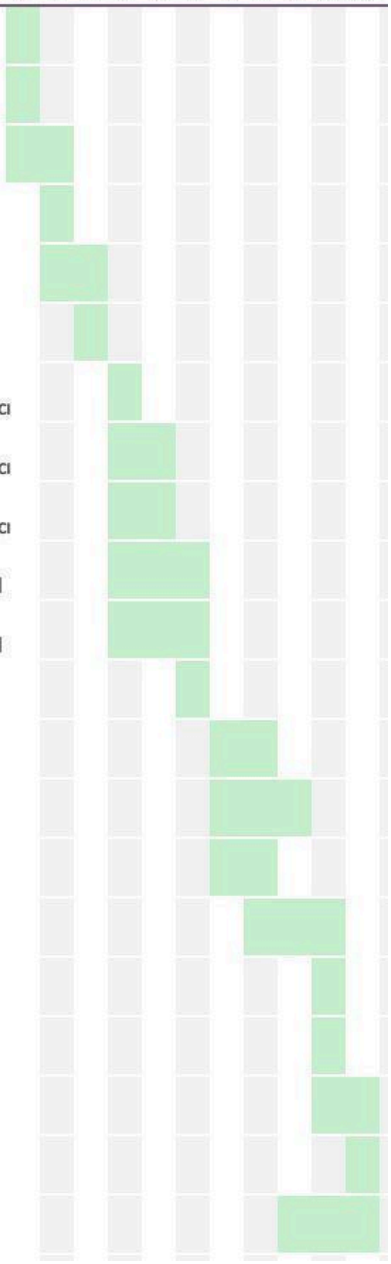
- When given enough training epochs and low learning rates, neural networks are outstanding at capturing intricate, non-linear relationships. They need to be carefully adjusted to prevent overfitting, though, and they are computationally demanding.
- Although they have trouble with extremely non-linear data, simpler models like SVMs and logistic regression can perform better than neural networks in low-epoch situations or when computing resources are scarce.
- By addressing class imbalance, data preprocessing—in particular, oversampling—significantly enhanced performance across all models.

This project emphasized the necessity of selecting models and hyperparameters according to the particular dataset and project requirements. To further increase classification accuracy and robustness, future research could be done to investigate other strategies like ensemble learning or sophisticated neural network architectures.

Gantt Chart For Member Contributions

Güzinsu Yaylacı – 22102007
Mehmet Oral – 22102156

ACTIVITY	PLAN START	PLAN DURATION	MEMBERS
Problem Research	1	1	Both
Dataset Research	1	1	Both
Method Research	1	2	Both
Proposed Dataset	2	1	Both
Propsed Methods	2	2	Both
Reporting (Proposal)	3	1	Both
Data Preprocessing	4	1	Güzinsu Yaylacı
Simulation Setup for Logistic Regression	4	2	Güzinsu Yaylacı
Simulation Result for Logistic Regression	4	2	Güzinsu Yaylacı
Simulation Setup for NN	4	3	Mehmet Oral
Simulation Result for NN	4	3	Mehmet Oral
Performance Discussion	6	1	Both
Reporting (First Report)	7	2	Both
Setup Improvement for Logistic Regression	7	3	Both
Setup Improvement for NN	7	2	Both
Simulation Setup of SVM	8	3	Both
Simulation Result of SVM	10	1	Both
Performance Discussion	10	1	Both
Revision and Possible Improvements	10	2	Both
Conclusion	11	1	Both
Reporting (Final Report)	9	3	Both



8. References

[1]Fedesoriano. “Stellar Classification Dataset - SDSS17.” *Kaggle*, 15 Jan. 2022, www.kaggle.com/datasets/fedesoriano/stellar-classification-dataset-sdss17.

[2]Majumder, Torsha. “Anomaly Detection and Classification of Astronomical Objects in the Era of Machine Learning.” *Astrobites*, 1 July 2024, astrobites.org/2024/07/07/guest-anomaly-detection/.

[3]Kavitha Subramani, Saranya R., Pushpavalli K., and G.B. Renuka. “Supervised Machine Learning Algorithm for Stellar Classification.” *Nanotechnology Perceptions*, Vol. 20, No. S10 (2024), pp. 499–509.

9. Appendix

Appendix A- Function Explanations

In Table 1, the common functions for all algorithms, such as loss and activation, are given.
Table 3: Common Functions for Both Algorithms:

Function Name	**Arguments**	**Returns**	**Purpose**
one_hot_representation	class_array (list/array): target class labels	one_hot_encoded (ndarray): one-hot encoded matrix label (list): list of one-hot vectors	Change the string labels to one hot encoded labels
loss	y_true (ndarray): true labels y_pred (ndarray): predicted probabilities	loss (float):	Calculate the loss
softmax	z (ndarray): Induced field	z_exp / np.sum(z_exp, axis=1, keepdims=True) (ndarray):	Compute the softmax of given induced field
accuracy_score	y_test (ndarray): y_pred (ndarray):	accuracy (float):	Calculate accuracy of the model
f1	y_test (ndarray): Test labels y_pred (ndarray): class_labels (list):	f1_score (float):	Calculate f1 score of the model
k_fold	X (dataframe): Input Y (series): Labels k (int): Number of folds shuffle (bool): Shuffling of data learning (float): Desired learning rate epoch (int): Epoch number batch_size (int): Desired batch size for gradient descent sgd (bool): Type of	accuracies (list): average_accuracy (float): f1_scores (list): average_f1_score (float):	Divide the dataset into k folds, train the model without the selected fold which iterates over k times. Make predictions to the selected fold. Calculate accuracy and f1 score for every fold and calculate their averages

	gradient descent		

Table 4: Functions of the LogisticRegression Class

Function Name	**Arguments**	**Purpose**
forward_pass	X (array): Training input	Calculating induced field and calculating softmax of the induced field to find output
fit	X (array): Training inputs Y (array): Training labels batch_size (int): Desired batch_size for gradient descent sgd (bool): Type of gradient descent	Train the model based on the input and labels. Call required functions to update the parameters
gradient	X (array): Training input Y (array): Training output	Calculate output with forward_pass(), calculate loss with loss() and add to a list, calculate gradients of the parameters
update_parameters	dw (array): Gradients of weights db (array): Gradients of biases	Update the parameters of model with respect to learning rate and gradients
predict	X (array): Test inputs	Make a prediction to test inputs with the model (predictions are one hot encoded like labels)

Table 5: Functions of the NN Class

Function Name	**Arguments**	**Purpose**
forward_pass	X (array): Output of layers i (int): Computation layer activation (str): activation function for hidden layers	Calculate the induced field, calculate the output based on induced field depending on the layer and the activation function
fit	X (array): Training input Y (array): Training labels batch_size (int): desired batch size sgd (bool): Desired type of gradient descent	Train the model based on the input and labels. Call required functions to update the parameters

feed_forward	X (array): Training input activation (str): activation function for hidden layers	Calculate the outputs and induced fields based on the input with the help of forward_pass()
update_parameters	delta (dictionary): Local gradients of each layer i (int): Number of hidden layers output (dictionary): Outputs of every layer (including input and output layer)	Update the weights and biases with local gradients and outputs of layers
predict	X (array): Test inputs	Make predictions with test inputs

Appendix B:
Data Preprocessing:

```
import numpy as np
import pandas as pd
import seaborn as sns
data = pd.read_csv("/content/drive/MyDrive/star_classification.csv", sep=',')
output = data['class']
input = data.drop(['class','rerun_ID'], axis=1)
input_max = input.max()
input_min = input.min()
input_normalized = (input - input_min) / (input_max - input_min)
df = pd.DataFrame(input_normalized)

# # Compute the correlation matrix
correlation_matrix = df.corr()

# # Visualize the correlation matrix using a heatmap
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.show()
dataset = pd.concat([input_normalized, output], axis=1)

# Shuffle the dataset (by shuffling rows)
shuffled_dataset = dataset.sample(frac=1, random_state=42).reset_index(drop=True)

# Calculate the index to split the dataset (90% for training and 10% for testing)
train_size = int(0.9 * len(shuffled_dataset))
```

```
# Split the dataset into training and testing sets
train_data = shuffled_dataset[:train_size]
test_data = shuffled_dataset[train_size:]

# Separate features and labels for training and testing
input_normalized = train_data.drop('class', axis=1)
output = train_data['class']

X_test = test_data.drop('class', axis=1)
Y_test = test_data['class']

import matplotlib.pyplot as plt

# Plot histogram of the classes
output.value_counts().plot(kind='bar', color='skyblue')
plt.title('Class Distribution Before Oversampling Minority Cases')
plt.xlabel('Classes')
plt.ylabel('Frequency')
plt.show()

# Check class distribution
class_counts = output.value_counts()
print(class_counts)

max_class_size = class_counts.max()
classes = class_counts.index

print(f"Majority Class Size: {max_class_size}")

# oversample
oversampled_data = []
# Loop through each class
for cls in classes:
  class_data = input_normalized[output == cls]  # Get data for the current class
  oversampled_class = class_data.sample(max_class_size, replace=True, random_state=42)  #
Oversample with replacement
```

```
  oversampled_data.append(oversampled_class)

# Combine all oversampled classes
oversampled_input = pd.concat(oversampled_data, axis=0)

# Create the corresponding labels
oversampled_output = pd.Series(
  [cls for cls in classes for _ in range(max_class_size)],
  name='class'
)



#shuffle
# Reset indices of oversampled data and labels
oversampled_input = oversampled_input.reset_index(drop=True)
oversampled_output = oversampled_output.reset_index(drop=True)

# Combine input and output
balanced_data = pd.concat([oversampled_input, oversampled_output], axis=1)

# Shuffle the dataset
balanced_data = balanced_data.sample(frac=1, random_state=42).reset_index(drop=True)
input = data.drop(['class'], axis=1)



import matplotlib.pyplot as plt

# Plot the updated class distribution histogram
balanced_data['class'].value_counts().plot(kind='bar', color='lightblue', edgecolor='black')
plt.title('Class Distribution After Oversampling')
plt.xlabel('Classes')
plt.ylabel('Frequency')
plt.xticks(rotation=0)
plt.show()
# Check the class distribution
print(balanced_data['class'].value_counts())
```

Güzinsu Yaylacı – 22102007
Mehmet Oral – 22102156

Logistic Regression:

```
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
def one_hot_representation(class_array):
  data = sorted(list(set(class_array)))

  num_classes = len(data)
  one_hot = np.eye(num_classes)
  label_to_index = {label: idx for idx, label in enumerate(data)}

  one_hot_encoded = one_hot[[label_to_index[label] for label in class_array]]



  label = np.split(one_hot,one_hot.shape[1],axis=0)



  return one_hot_encoded , label

def loss(y_true, y_pred):

  epsilon = 1e-9  # Small value to prevent log(0)

  y_pred = np.clip(y_pred, epsilon, 1. - epsilon)


  loss = -np.sum(y_true * np.log(y_pred)) / y_true.shape[0]
  return loss
def softmax(z):

  z_exp = np.exp(z - np.max(z, axis=1, keepdims=True))
  return z_exp / np.sum(z_exp, axis=1, keepdims=True)
```

```
def accuracy_score(y_test, y_pred):
  if len(y_test) != len(y_pred):
     raise ValueError("The length of y_test and y_pred must be the same.")
  y_test_class = np.argmax(y_test, axis=1)
  y_pred_class = np.argmax(y_pred, axis=1)
  correct_predictions = np.sum(y_test_class == y_pred_class)

  accuracy = correct_predictions / len(y_test)

  return accuracy
def compute_confusion_matrix(y_test , y_pred):

  Y_pred_classes = np.argmax(y_pred, axis=1)
  Y_test_classes = np.argmax(y_test, axis=1)
  conf_matrix = confusion_matrix(Y_test_classes, Y_pred_classes)

  disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=["Galaxy","Star",
"QSO"])
  disp.plot(cmap="Blues")
  plt.title("Confusion Matrix for Logistic Regression")
  plt.show()


  return conf_matrix

def f1(y_test, y_pred, class_labels):
  f1_scores = []

  for label in class_labels:
     TP = np.sum((y_test == label) & (y_pred == label))
     FP = np.sum((y_test != label) & (y_pred == label))
     FN = np.sum((y_test == label) & (y_pred != label))

     precision = TP / (TP + FP) if (TP + FP) > 0 else 0
     recall = TP / (TP + FN) if (TP + FN) > 0 else 0

     f1_class = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0
```

```
    f1_scores.append(f1_class)

  return np.mean(f1_scores)

class LogisticRegression:
  def __init__(self, learning_rate=0.001, epoch=1000, num_classes=3, batch_size = 1):
    self.lr = learning_rate
    self.epoch = epoch
    self.classes = num_classes
    self.parameters = {}
    self.losses = []
    self.batch_size = batch_size


  def fit(self, X, y, batch_size=32, SGD=False ):

    n_samples, n_features = X.shape

    self.parameters["W1"] = np.random.randn(n_features, self.classes) * 0.01
    self.parameters["b1"] = np.zeros((1, self.classes))

    for epoch in range(self.epoch):
      if SGD:
        indices = np.random.permutation(n_samples)
        X_shuffled = X[indices]
        y_shuffled = y[indices]
        n_batch= X_shuffled.shape[0] // batch_size

        for i in range(n_batch):
          xi = X_shuffled[(i*batch_size):((i+1)*batch_size)]
          yi = y_shuffled[(i*batch_size):((i+1)*batch_size)]
          dw, db = self.gradient(xi,yi)
          self.update_parameters(dw,db)
        if X.shape[0] % batch_size != 0:
          xi = X_shuffled[(n_batch*batch_size)::]
          yi = y_shuffled[(n_batch*batch_size)::]
          dw, db = self.gradient(xi,yi)
```

```
            self.update_parameters(dw,db)
        else:
            X_shuffled = X
            y_shuffled = y

            n_batch= X_shuffled.shape[0] // batch_size

            for i in range(n_batch):
                xi = X_shuffled[(i*batch_size):((i+1)*batch_size)]
                yi = y_shuffled[(i*batch_size):((i+1)*batch_size)]
                dw, db = self.gradient(xi,yi)
                self.update_parameters(dw,db)
            if X.shape[0] % batch_size != 0:
                xi = X_shuffled[(n_batch*batch_size)::]
                yi = y_shuffled[(n_batch*batch_size)::]

                dw, db = self.gradient(xi,yi)
                self.update_parameters(dw,db)

def forward_pass(self,X):

    z = np.dot(X, self.parameters["W1"]) + self.parameters["b1"]

    A = softmax(z)
    return A



def gradient(self, X, Y):
    A = self.forward_pass(X)
    self.losses.append(loss(Y, A))
    dz = A - Y

    dw = (1 / X.shape[0]) * np.dot(X.T, dz)
    db = (1 / X.shape[0]) * np.sum(dz)
    return dw, db
def update_parameters(self,dw,db):
```

```
        self.parameters["W1"] = self.parameters["W1"] - self.lr * dw
        self.parameters["b1"] = self.parameters["b1"] - self.lr * db

    def predict(self, X):

        y_prediction = np.zeros((X.shape[0],self.classes))

        y_hat = np.dot(X, self.parameters["W1"]) + self.parameters["b1"]
        y_pred = softmax(y_hat)

        max_i = np.argmax(y_pred, axis=1)



        y_prediction[np.arange(len(y_pred)),max_i] = 1

        return np.array(y_prediction)

def k_fold(X, Y, k=5, shuffle=False,learning = (0.01, 0.001), epoch = (50, 100, 150) , batch_size=80000,
sgd = (False, True)):
    if shuffle:
        indices = np.random.permutation(len(X))

        X_shuffled = X.iloc[indices]
        Y_shuffled = Y.iloc[indices]



    else:
        X_shuffled = X
        Y_shuffled = Y
    Y_shuffled_ohr, oh_labels = one_hot_representation(Y_shuffled)

    fold_size = len(X) // k
    accuracies = []
    f1_scores = []
    cm = []
    model_accuracies = []
    model_f1_scores = []
```

```
models = []
#model_cm = []
for m in range(len(epoch)):
    epochs = epoch[m]

    for j in range(len(learning)):
        learning_rate = learning[j]

        for l in range(len(sgd)):
            sgd_bool = sgd[l]

            for i in range(k):

                test_start = i * fold_size

                test_end = (i + 1) * fold_size if i != k - 1 else len(X)

                X_train = np.concatenate([X_shuffled[:test_start], X_shuffled[test_end:]], axis=0)
                Y_train = np.concatenate([Y_shuffled_ohr[:test_start], Y_shuffled_ohr[test_end:]], axis=0)
                X_valid = X_shuffled[test_start:test_end]
                Y_valid = Y_shuffled_ohr[test_start:test_end]



                model = LogisticRegression(learning_rate = learning_rate, epoch = epochs,num_classes = 3)

                model.fit(X_train, Y_train, batch_size, SGD= sgd_bool)

                Y_pred = model.predict(X_valid)

                accuracy = accuracy_score(Y_valid, Y_pred)
                accuracies.append(accuracy)
                f1_score = f1(Y_valid, Y_pred, oh_labels)
                f1_scores.append(f1_score)
            average_accuracy = np.mean(accuracies)
            average_f1_score = np.mean(f1_scores)
            print("Logistic Regression: ")
```

```
            print("For learning rate " + str(learning_rate) + " and epoch " + str(epochs) + " and sgd " +
str(sgd_bool))
            print(f"Validation Accuracy: {average_accuracy:.2f}")
            print(f"Validation F1 Score: {average_f1_score:.2f}\n")

  model_accuracies.append(average_accuracy)
  model_f1_scores.append(average_f1_score)
  models.append(model)
  #model_cm.append(cm)
  return  model_accuracies, model_f1_scores, models

data = balanced_data

output = data['class']

input = data.drop(['class'], axis=1)



input_max = input.max()

input_min = input.min()

input_normalized = (input - input_min) / (input_max - input_min)
Y_test_ohr, oh_lab = one_hot_representation(Y_test)

Acc, F1, models = k_fold(input_normalized, output, shuffle=True, k=5, learning=(0.01, 0.001) ,epoch=(5,
10, 15), batch_size=1, sgd = (False, True))
best = np.argmax[F1]
best_model = models[best]
best_pred = best_model.predict(X_test)
accuracy = accuracy_score(Y_test_ohr, best_pred)
f1_score = f1(Y_test_ohr, best_pred, oh_lab)

cm = compute_confusion_matrix(Y_test_ohr, best_pred)

print("For Best Model of Logistic Regression: ")
print("Test Accuracy is " + str(accuracy))
```

```
print("Test F1 Score is " + str(f1_score))
```

Neural Networks:

```
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

def one_hot_representation(class_array):
  data = sorted(list(set(class_array)))
  num_classes = len(data)
  one_hot = np.eye(num_classes)
  label_to_index = {label: idx for idx, label in enumerate(data)}

  one_hot_encoded = one_hot[[label_to_index[label] for label in class_array]]


  label = np.split(one_hot,one_hot.shape[1],axis=0)

  return one_hot_encoded , label
def sigmoid(x):
  return 1 / (1 + np.exp(-x))


  return one_hot_encoded , label
def activation_derivative(z, activation):
  if activation == "relu":

    derivative = (z > 0).astype(float)
  elif activation == "sigmoid":
    derivative = sigmoid(z)*(1-sigmoid(z))
  else:
```

```
        derivative = 0
    return np.diag(derivative)
def loss(y_true, output , h_layers):

    epsilon = 1e-9
    y_pred = np.clip(output["A" + str(h_layers+1)], epsilon, 1. - epsilon)



    loss = -np.sum(np.dot(y_true.T , np.log(y_pred))) / y_true.shape[0]
    return loss
def compute_confusion_matrix(y_test , y_pred):

    Y_pred_classes = np.argmax(y_pred, axis=1)
    Y_test_classes = np.argmax(y_test, axis=1)
    conf_matrix = confusion_matrix(Y_test_classes, Y_pred_classes)

    disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=["Galaxy", "Star",
"QSO"])
    disp.plot(cmap="Blues")
    plt.title("Confusion Matrix for NN")
    plt.show()

    return conf_matrix
def softmax(z):

    z_exp = np.exp(z - np.max(z, axis=1, keepdims=True))
    return z_exp / np.sum(z_exp, axis=1, keepdims=True)
def accuracy_score(y_test, y_pred):
    if len(y_test) != len(y_pred):
        raise ValueError("The length of y_test and y_pred must be the same.")
    y_test_class = np.argmax(y_test, axis=1)
    y_pred_class = np.argmax(y_pred, axis=1)
    correct_predictions = np.sum(y_test_class == y_pred_class)

    accuracy = correct_predictions / len(y_test)

    return accuracy
```

```
def f1(y_test, y_pred, class_labels):
  f1_scores = []

  for label in class_labels:
     TP = np.sum((y_test == label) & (y_pred == label))
     FP = np.sum((y_test != label) & (y_pred == label))
     FN = np.sum((y_test == label) & (y_pred != label))

     precision = TP / (TP + FP) if (TP + FP) > 0 else 0
     recall = TP / (TP + FN) if (TP + FN) > 0 else 0

     f1_class = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0
     f1_scores.append(f1_class)

  return np.mean(f1_scores)

class NN:
  def __init__(self, learning_rate=0.001, epoch=1000, num_classes= 3,num_hidden=[10], activation =
"relu"):
     self.lr = learning_rate
     self.epoch = epoch
     self.classes = num_classes
     self.parameters = {}
     self.losses = []
     self.h_layers = len(num_hidden)
     self.hidden_n = num_hidden
     self.activation = activation



  def fit(self, X, y, batch_size=32, SGD=False):
     n_samples, n_features = X.shape
```

```
    self.parameters["W1"] = np.random.randn(n_features, self.hidden_n[0]) * 0.01
    self.parameters["b1"] = np.zeros((1, self.hidden_n[0]))
    if self.h_layers > 1:
        for l in range(1, self.h_layers):

            self.parameters["W" + str(l+1)] = np.random.randn(self.hidden_n[l-1], self.hidden_n[l]) * 0.01
            self.parameters["b" + str(l+1)] = np.zeros((1, self.hidden_n[l]))
    self.parameters["W"+ str(self.h_layers+1)] = np.random.randn(self.hidden_n[-1], self.classes) * 0.01
    self.parameters["b"+ str(self.h_layers+1)] = np.zeros((1, self.classes))


    for epoch in range(self.epoch):
        if SGD:
            indices = np.random.permutation(n_samples)
            X_shuffled = X[indices]
            y_shuffled = y[indices]
            n_batch= X_shuffled.shape[0] // batch_size
            for i in range(n_batch):
                xi = X_shuffled[(i*batch_size):((i+1)*batch_size)]
                yi = y_shuffled[(i*batch_size):((i+1)*batch_size)]
                delta = {}

                output, ind_field = self.feed_forward(xi,self.activation)
                self.losses.append(loss(yi, output, self.h_layers))

                delta["d" + str(self.h_layers + 1)] = output["A" + str(self.h_layers + 1)] - yi

                for i in range(self.h_layers, 0, -1):
                    delta["d" + str(i)] = np.dot(delta["d" + str(i + 1)],  self.parameters["W"+ str(i+1)].T) *
activation_derivative(ind_field["z" + str(i)], self.activation)
                self.update_parameters(delta,self.h_layers, output)
            if X.shape[0] % batch_size != 0:
                xi = X_shuffled[(n_batch*batch_size)::]
                yi = y_shuffled[(n_batch*batch_size)::]
                delta = {}
```

```
                output, ind_field = self.feed_forward(xi,self.activation)
                self.losses.append(loss(yi, output,self.h_layers))

                delta["d" + str(self.h_layers + 1)] = output["A" + str(self.h_layers + 1)]- yi
                for i in range(self.h_layers, 0, -1):
                    delta["d" + str(i)] = np.dot(delta["d" + str(i + 1)],  self.parameters["W"+ str(i+1)].T) *
activation_derivative(ind_field["z" + str(i)], self.activation)
                self.update_parameters(delta,self.h_layers, output)
            else:
                X_shuffled = X
                y_shuffled = y
                n_batch= X_shuffled.shape[0] // batch_size
                for i in range(n_batch):
                    xi = X_shuffled[(i*batch_size):((i+1)*batch_size)]
                    yi = y_shuffled[(i*batch_size):((i+1)*batch_size)]
                    delta = {}

                    output, ind_field = self.feed_forward(xi,self.activation)
                    self.losses.append(loss(yi, output, self.h_layers))

                    delta["d" + str(self.h_layers + 1)] = output["A" + str(self.h_layers + 1)] - yi

                    for i in range(self.h_layers, 0, -1):
                        delta["d" + str(i)] = np.dot(delta["d" + str(i + 1)],  self.parameters["W"+ str(i+1)].T) *
activation_derivative(ind_field["z" + str(i)], self.activation)

                    self.update_parameters(delta,self.h_layers, output)
                if X.shape[0] % batch_size != 0:
                    xi = X_shuffled[(n_batch*batch_size)::]
                    yi = y_shuffled[(n_batch*batch_size)::]
                    delta = {}

                    output, ind_field = self.feed_forward(xi,self.activation)
                    self.losses.append(loss(yi, output, self.h_layers))

                    delta["d" + str(self.h_layers + 1)] = output["A" + str(self.h_layers + 1)] - yi
```

```
            for i in range(self.h_layers, 0, -1):
                delta["d" + str(i)] = np.dot(delta["d" + str(i + 1)],  self.parameters["W"+ str(i+1)].T) *
activation_derivative(ind_field["z" + str(i)], self.activation)

            self.update_parameters(delta,self.h_layers)

  def forward_pass(self,X,i,activation):

    if i  == self.h_layers:

      z = np.dot(X, self.parameters["W" + str(i+1)]) + self.parameters["b" + str(i+1)]
      A = softmax(z)

    else:


      z = np.dot(X, self.parameters["W" + str(i+1)]) + self.parameters["b" + str(i+1)]
      if activation == "relu":
        A = np.maximum(0,z)
      elif activation == "sigmoid":
        A = sigmoid(z)
    return A , z


  def feed_forward(self, X , activation):
    output = {}
    ind_field = {}
    output["A" + str(0)] = X

    for i in range(self.h_layers+1):
      output["A" + str(i+1)], ind_field["z" + str(i+1)] = self.forward_pass(output["A" + str(i)] ,i,
activation)

    return output , ind_field

  def update_parameters(self, delta , i, output):
    for l in range(i + 1, 0, -1):
```

```
        self.parameters['W' + str(l)] = self.parameters['W' + str(l)]  - self.lr * np.dot(output["A" + str(l-1)].T , delta["d" + str(l)] )
        self.parameters['b' + str(l)]  = self.parameters['b' + str(l)]  - self.lr * delta["d" + str(l)]

  def predict(self, X):

      y_prediction = np.zeros((X.shape[0],self.classes))
      output , _ = self.feed_forward(X ,self.activation)

      max_i = np.argmax(output['A'+ str(self.h_layers + 1)], axis=1)


      y_prediction[np.arange(len(output['A'+ str(self.h_layers + 1)])),max_i] = 1

      return np.array(y_prediction)

def k_fold(X, Y, k=5, shuffle=False, learning=0.001, epoch=(5, 10), batch_size=80000, sgd=True, hidden_layers=([5], [10], [10, 5]), activations=("relu", "sigmoid")):


  if shuffle:
      indices = np.random.permutation(len(X))

      X_shuffled = X.iloc[indices]
      Y_shuffled = Y.iloc[indices]

  else:
      X_shuffled = X
      Y_shuffled = Y
  Y_shuffled_ohr, oh_labels = one_hot_representation(Y_shuffled)

  fold_size = len(X) // k
  accuracies = []
  f1_scores = []
  cm = []
  model_accuracies = []
  model_f1_scores = []
```

```
  models = []
  for m in range(len(epoch)):
     epochs  = epoch[m]
     for o in range(len(hidden_layers)):
        hidden_layer = hidden_layers[o]
        for p in range(len(activations)):
           activation_function = activations[p]
           for i in range(k):
              test_start = i * fold_size
              test_end = (i + 1) * fold_size if i != k - 1 else len(X)

              X_train = np.concatenate([X_shuffled[:test_start], X_shuffled[test_end:]], axis=0)
              Y_train = np.concatenate([Y_shuffled_ohr[:test_start], Y_shuffled_ohr[test_end:]], axis=0)
              X_test = X_shuffled[test_start:test_end]
              Y_test = Y_shuffled_ohr[test_start:test_end]



              model = NN(learning_rate=learning, epoch=epochs, num_classes=
3,num_hidden=hidden_layer, activation = activation_function)

              batch_s = min(batch_size, X_train.shape[0])

              model.fit(X_train, Y_train, batch_s, SGD= sgd)

              Y_pred = model.predict(X_test)

              accuracy = accuracy_score(Y_test, Y_pred)
              accuracies.append(accuracy)

              f1_score = f1(Y_test, Y_pred, oh_labels)
              f1_scores.append(f1_score)
              #cm.append(compute_confusion_matrix(Y_test, Y_pred))
        average_accuracy = np.mean(accuracies)
        average_f1_score = np.mean(f1_scores)
        print("Neural Networks: ")
        print("For learning rate " + str(learning) + " and epoch " + str(epochs)  + " and hidden layers " +
str(hidden_layer) + " and activation " + str(activation_function))
```

```
        print("Validation Accuracy: " + str(average_accuracy))
        print("Validation F1 Score: " + str(average_f1_score) + "\n")

        model_accuracies.append(average_accuracy)
        model_f1_scores.append(average_f1_score)
        models.append(model)
        #model_cm.append(cm)
    return model_accuracies, model_f1_scores, models

data = balanced_data

output = data['class']

input = data.drop(['class'], axis=1)


input_max = input.max()

input_min = input.min()

input_normalized = (input - input_min) / (input_max - input_min)

k_fold(input_normalized, output, k=5, shuffle=False, learning=0.001, epoch=(5, 10, 15), batch_size=1,
sgd=True, hidden_layers=([10, 5],[10],[5]), activations = ("relu", "sigmoid"))

Y_test_ohr, oh_lab = one_hot_representation(Y_test)

best = np.argmax(F1)
best_model = models[best]
best_predictions = best_model.predict(X_test)
accuracy = accuracy_score(Y_test_ohr, best_predictions)
f1_score = f1(Y_test_ohr, best_pred, oh_lab)
cm = compute_confusion_matrix(Y_test_ohr, best_pred)

print("For Best Model of NN: ")
print("Test Accuracy is " + str(accuracy))
```

```
print("Test F1 Score is " + str(f1_score))
```

SVM:

```
class SVM:
        def _init_(self, learning_rate=0.001, epoch=10, num_classes=3, kernel = "linear",
regularization=0):
        self.lr = learning_rate
        self.epoch = epoch
        self.classes = num_classes
        self.parameters={}
        self.losses = {}
        self.kernel = kernel
        self.regularization = regularization
        def kernel_function(self, X, X2=None):
        if self.kernel == 'linear':
        return X
        elif self.kernel == 'rbf':
        gamma = 0.5
        sq_dists = np.sum(X*2, axis=1).reshape(-1, 1) + np.sum(X2*2, axis=1) - 2 * np.dot(X, X2.T)
        return np.exp(-gamma * sq_dists)
        elif self.kernel == 'polynomial':
        degree = 2
        coef0 = 2
        return (np.dot(X, X2.T) + coef0) ** degree
        else:
        raise ValueError("Kernel not supported")

        def forward_pass(self, X):
        # Use the kernel function to calculate the decision function
        K = self.kernel_function(X)
        z = np.dot(K, self.parameters["W"]) + self.parameters["b"]
        A = softmax(z)
        return A

        def fit(self, X, y):


        n_samples, n_features = X.shape

        # for i in range(n_features):

        self.parameters["W"] = np.random.randn(n_features,1) * 0.01
```

```
        self.parameters["b"] = np.zeros((1,1))


        for _ in range(self.epoch):
        for j in range(n_samples):

        A = X[j].reshape(n_features,1)

        prediction =(np.dot(A.T, self.parameters["W"]) + self.parameters["b"])

        margin = y[j]* prediction
        # print((np.dot(X[j]., self.parameters["W"])).shape)
        if margin >= 1:
                dw = 2 * self.regularization * self.parameters["W"]
                db = 0
        else:
                dw = 2 * self.regularization * self.parameters["W"] - y[j] * A
                db = y[j]
        self.parameters["W"] = self.parameters["W"] - self.lr * dw
        self.parameters["b"] = self.parameters["b"] - self.lr * db

        def predict(self, X):



        y_hat = np.dot(X, self.parameters["W"]) + self.parameters["b"]
        A = np.sign(y_hat)

        return A
def compute_confusion_matrix(y_test , y_pred):

        Y_pred_classes = np.argmax(y_pred, axis=1)
        Y_test_classes = np.argmax(y_test, axis=1)

        conf_matrix = confusion_matrix(Y_test_classes, Y_pred_classes)

        disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=["Galaxy",
"Star", "QSO"])
        disp.plot(cmap="Blues")
        plt.title("Confusion Matrix for SVM")
        plt.show()

        return conf_matrix
def k_fold(X, Y, k=5, shuffle=False,learning = 0.001,epoch = 1000 ):

        if shuffle:
        indices = np.random.permutation(len(X))

        X_shuffled = X.iloc[indices]
        Y_shuffled = Y.iloc[indices]
```

```
        else:
        X_shuffled = X
        Y_shuffled = Y
        Y_shuffled_ohr, oh_labels = one_hot_representation(Y_shuffled) # One hot representation of
classes

        # Step 2: Split data into k folds
        fold_size = len(X) // k
        accuracies = []  # List to store accuracy for each fold
        f1_scores = []
        cm = []
        for i in range(k):
        # Step 3: Define the test set for this fold
        test_start = i * fold_size
        test_end = (i + 1) * fold_size if i != k - 1 else len(X)

        # Split data into training and test sets
        X_train = np.concatenate([X_shuffled[:test_start], X_shuffled[test_end:]], axis=0)
        Y_train = np.concatenate([Y_shuffled_ohr[:test_start], Y_shuffled_ohr[test_end:]], axis=0)
        X_test = X_shuffled[test_start:test_end]
        Y_test = Y_shuffled_ohr[test_start:test_end]


        # Step 4: Train the model on the training set
        model = MultiClassSVM(len(oh_labels)
        # model(learning , iterations , num_class)
        model.fit(X_train, Y_train)

        # Step 5: Make predictions on the test set
        Y_pred = model.predict(X_test)

        # Step 6: Calculate accuracy for this fold
        accuracy = accuracy_score(Y_test, Y_pred)
        accuracies.append(accuracy)
        f1_score = f1(Y_test, Y_pred, oh_labels)
        f1_scores.append(f1_score)
        cm.append(compute_confusion_matrix(Y_test, Y_pred))
        # Step 7: Calculate the average accuracy across all folds
        average_accuracy = np.mean(accuracies)
        average_f1_score = np.mean(f1_scores)

        return accuracies, average_accuracy, f1_scores, average_f1_score,cm


data = balanced_data

output = data['class']
```

```
input = data.drop(['class'], axis=1)
# df = pd.DataFrame(input)

# # Compute the correlation matrix
# correlation_matrix = df.corr()

# # Visualize the correlation matrix using a heatmap
# sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
# plt.show()

input_max = input.max()

input_min = input.min()

input_normalized = (input - input_min) / (input_max - input_min)


Acc, Av_acc, F1, Av_f1, cm = k_fold(input_normalized, output, shuffle=True,learning = 0.001
,epoch=10)
print("For NN: ")
print("Accuracies are " + str(Acc))
print("Average Accuracy is " +str(Av_acc))
print("For NN: ")
print("F1 Scores are " + str(F1))
print("Average F1 Score is " + str(Av_f1))
```