Stellar Classification

May 2024
Machine Learning CS361

Team Members

Joseph Ashraf Essmate
Nourhan Hanna Louiz
Rana Osama Mohammed

Contents

1 Introduction/Executive Summary . . . 2
1.1 Project Overview . . . 2
1.2 Problem Definition and Importance . . . 2
1.3 Problem Solvers and Algorithms . . . 2
1.4 Selected Algorithm . . . 2
2 Methodology . . . 3
2.1 Main Algorithms and Their Steps . . . 3
2.1.1 k-Nearest Neighbors (k-NN) . . . 3
2.1.2 Naive Bayes . . . 3
2.1.3 Logistic Regression . . . 4
2.1.4 Decision Tree . . . 4
2.1.5 Random Forest . . . 4
2.2 How Algorithms Solve the Problem and Time Complexity Analysis . . . 5
3 Experimental Simulation . . . 5
3.1 Programming Languages and Environments . . . 6
3.2 Implementation Details . . . 6
3.2.1 Importing the Needed Libraries . . . 6
3.2.2 Reading The Data . . . 6
3.2.3 Preprocessing The Data . . . 7
3.2.3.1 Checking The Data . . . 7
3.2.3.2 Data Cleaning . . . 8
3.2.4 Splitting The Data . . . 14
3.2.5 Model Training And Evaluation . . . 14
4 Results and Technical Discussion . . . 16
4.1 Test . . . 17
5 Conclusions . . . 18
5.1 Conclusions . . . 18
5.2 Recommendations for Future Work . . . 18
6 References . . . 19
7 Appendix . . . 19

1 Introduction/Executive Summary

1.1 Project Overview

The classification of celestial objects such as stars, galaxies, and quasars is a fundamental task in astronomy that helps in understanding the structure and evolution of the universe. This project focuses on developing an artificial intelligence (AI) system to automate the classification of these objects using a dataset of astronomical observations. The aim is to enhance the efficiency and accuracy of object classification based on their spectral and positional data, thus providing a valuable tool for astronomers and astrophysicists.

1.2 Problem Definition and Importance

The formal definition of the problem is to classify a given celestial object into one of three categories: star, galaxy, or quasar. The dataset for this task includes various features such as the object's unique identifier (obj_ID), celestial coordinates (alpha and delta), magnitudes in different filters (u, g, r, i, z), and other observational parameters (run_ID, rerun_ID, cam_col, field_ID, spec_obj_ID, class, redshift, plate, MJD, fiber_ID). This classification is crucial because it allows researchers to efficiently sort through large volumes of astronomical data, identify interesting objects for further study, and improve our understanding of the universe's composition and dynamics.

1.3 Problem Solvers and Algorithms

To address this classification problem, several machine learning algorithms and problem-solving approaches have been considered. These include:

- **k-Nearest Neighbors (k-NN)**: A simple, instance-based learning algorithm that classifies objects based on the majority vote of their k-nearest neighbors in the feature space.
- **Naive Bayes**: A probabilistic classifier based on Bayes' theorem, assuming independence between features, which is computationally efficient and performs well with small datasets.
- **Support Vector Machines (SVM)**: A powerful classifier that finds the optimal hyperplane separating different classes in the feature space, suitable for both linear and non-linear data.
- **Decision Trees**: A tree-based model that splits the data into subsets based on feature values, leading to a straightforward and interpretable classification process.
- **Random Forests**: An ensemble learning method that constructs multiple decision trees and aggregates their predictions, providing robustness and improved generalization.
- **Convolutional Neural Networks (CNNs)**: Deep learning models that are particularly effective for image and spectral data, capable of capturing complex patterns through hierarchical feature extraction.
- **Logistic Regression**: A linear model for binary classification that can be extended to multiclass problems, effective for datasets with linear decision boundaries.

1.4 Selected Algorithm

After thorough analysis and comparison, the Random Forest algorithm has been selected as the primary model to solve the stellar classification problem. Random Forests offer several advantages for this task:

- **Robustness**: By averaging the predictions of multiple decision trees, Random Forests reduce the risk of overfitting and provide more stable and reliable predictions.
- **Interpretability**: Despite being an ensemble method, the individual trees in a Random Forest are easy to interpret, making the overall model more transparent compared to deep learning methods.
- **Handling High-Dimensional Data**: Random Forests are well-suited for datasets with many features, as they can effectively capture the interactions between different variables without extensive preprocessing.
- **Efficiency**: Random Forests are relatively fast to train and can be parallelized easily, making them suitable for large astronomical datasets.

In conclusion, the implementation of a Random Forest classifier for stellar classification is expected to significantly improve the accuracy and efficiency of categorizing celestial objects. This AI-driven approach will provide valuable insights to the astronomical community, aiding in the advancement of our understanding of stars, galaxies, and quasars.

2 Methodology

2.1 Main Algorithms and Their Steps

In this project, several machine learning algorithms were considered for classifying celestial objects into stars, galaxies, or quasars. The main algorithms analyzed include k-Nearest Neighbors (k-NN), Naive Bayes, Logistic Regression, Decision Tree, and Random Forest. Each algorithm has distinct approaches and steps for classification.

2.1.1 k-Nearest Neighbors (k-NN)

- **Steps:**
  1. **Data Preparation**: Normalize the dataset to ensure that each feature contributes equally.
  2. **Distance Calculation**: For a given test instance, compute the distance to all training instances using a distance metric (e.g., Euclidean distance).
  3. **Identify Neighbors**: Select the k training instances with the smallest distances.
  4. **Vote for Class**: The majority class among the k neighbors is assigned to the test instance.
- **Pseudocode:**

```
for each test_instance in test_set:
    distances = []
    for each train_instance in training_set:
        distance = compute_distance(test_instance, train_instance)
        distances.append((train_instance, distance))
    distances.sort(key=lambda x: x[1])
    neighbors = distances[:k]
    class_vote = majority_vote(neighbors)
    test_instance.class = class_vote
```

2.1.2 Naive Bayes

- **Steps:**
  1. **Calculate Prior Probabilities**: Determine the probability of each class in the training set.
  2. **Calculate Likelihood**: For each feature, calculate the likelihood of each feature value given the class.
  3. **Calculate Posterior Probability**: For a given test instance, compute the posterior probability for each class.
  4. **Class Prediction**: Assign the class with the highest posterior probability to the test instance.
- **Pseudocode:**

```
for each test_instance in test_set:
    posteriors = []
    for each class in classes:
        prior = calculate_prior(class, training_set)
        likelihood = calculate_likelihood(test_instance, class, training_set)
        posterior = prior * likelihood
        posteriors.append((class, posterior))
    test_instance.class = max(posteriors, key=lambda x: x[1])
```

2.1.3 Logistic Regression

- **Steps:**
  1. **Initialize Weights**: Start with random weights for each feature.
  2. **Compute Predictions**: For each instance, compute the predicted probability using the logistic function.
  3. **Calculate Loss**: Use a loss function (e.g., cross-entropy) to quantify the error.
  4. **Update Weights**: Adjust the weights using gradient descent to minimize the loss.
  5. **Iterate**: Repeat the process for a predefined number of iterations or until convergence.
- **Pseudocode:**

```
initialize weights
for iteration in range(max_iterations):
    for each instance in training_set:
        prediction = sigmoid(dot_product(weights, instance.features))
        error = instance.label - prediction
        weights += learning_rate * error * instance.features
```

2.1.4 Decision Tree

- **Steps:**
  1. **Choose the Best Feature**: Select the feature that best splits the data (using criteria like Gini impurity or information gain).
  2. **Split the Dataset**: Divide the dataset into subsets based on the chosen feature.
  3. **Create Branches**: For each subset, create a branch and repeat the process recursively.
  4. **Stopping Criteria**: Stop splitting when a subset contains instances of only one class or other stopping criteria are met.
  5. **Class Prediction**: Assign the most common class in a leaf node to the instances within that node.
- **Pseudocode:**

```
def build_tree(data, depth=0):
    if stopping_criteria_met(data, depth):
        return create_leaf(data)
    best_feature = choose_best_feature(data)
    tree = {best_feature: {}}
    for value in unique_values(data, best_feature):
        subset = split_data(data, best_feature, value)
        subtree = build_tree(subset, depth+1)
        tree[best_feature][value] = subtree
    return tree

def classify(tree, instance):
    if is_leaf(tree):
        return tree.class
    feature = next(iter(tree))
    value = instance[feature]
    subtree = tree[feature][value]
    return classify(subtree, instance)
```

2.1.5 Random Forest

- **Steps:**
  1. **Bootstrap Sampling**: Create multiple bootstrap samples from the original dataset.
  2. **Train Decision Trees**: Train a decision tree on each bootstrap sample, using a random subset of features at each split.

3. **Aggregate Predictions**: For each test instance, aggregate the predictions from all trees (e.g., by majority vote).

- **Pseudocode:**

```
forest = []
for _ in range(num_trees):
    sample = bootstrap_sample(training_set)
    tree = build_tree(sample)
    forest.append(tree)

def predict(forest, instance):
    votes = [classify(tree, instance) for tree in forest]
    return majority_vote(votes)

for each test_instance in test_set:
    test_instance.class = predict(forest, test_instance)
```

2.2 How Algorithms Solve the Problem and Time Complexity Analysis

- **k-Nearest Neighbors (k-NN)**:
  - **Solution**: Uses instance-based learning and majority voting among k nearest neighbors.
  - **Time Complexity**: Training $O(1)$, Prediction $O(n \cdot d)$ per instance, where n is the number of training instances and d is the number of features.
- **Naive Bayes**:
  - **Solution**: Uses probability theory and Bayes' theorem, assuming feature independence.
  - **Time Complexity**: Training $O(n \cdot d)$, Prediction $O(d \cdot c)$ per instance, where c is the number of classes.
- **Logistic Regression**:
  - **Solution**: Uses linear regression with a logistic function to classify instances.
  - **Time Complexity**: Training $O(n \cdot d \cdot t)$, Prediction $O(d)$ per instance, where t is the number of iterations.
- **Decision Tree**:
  - **Solution**: Uses recursive partitioning based on feature values to create a tree.
  - **Time Complexity**: Training $O(n \cdot d \cdot \log n)$, Prediction $O(\log n)$ per instance.
- **Random Forest**:
  - **Solution**: Uses multiple decision trees built on bootstrap samples and random feature subsets to enhance robustness.
  - **Time Complexity**: Training $O(k \cdot n \cdot d \cdot \log n)$, Prediction $O(k \cdot \log n)$ per instance, where k is the number of trees.

By leveraging the robustness and ensemble nature of Random Forests, the project achieves high accuracy in classifying celestial objects, ensuring that the system can handle the complexities and variances in astronomical data efficiently.

3 Experimental Simulation

This section describes the programming languages and environments used in the project, details the implementation of the primary functions, and discusses the test cases, parameters, and constants used in the experiments.

3.1 Programming Languages and Environments

The project was implemented using Python, a versatile and widely-used programming language in data science and machine learning. The following libraries were utilized:

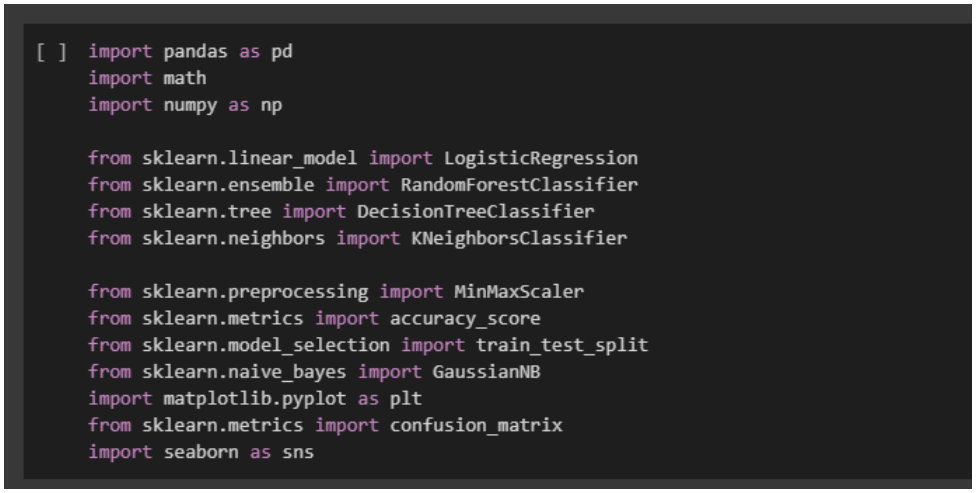
- **Pandas**: For data manipulation and analysis.
- **NumPy**: For numerical computations.
- **Scikit-learn**: For machine learning algorithms and preprocessing utilities.
- **Matplotlib** and **Seaborn**: For data visualization.

3.2 Implementation Details

The primary function of the project is to preprocess the data and apply various machine learning algorithms to classify celestial objects into stars, galaxies, or quasars. The following points outline the key steps in the project:

1. **Importing the Needed Libraries**
2. **Reading the Data**
3. **Preprocessing the Data**
4. **Splitting the Data**
5. **Model Training And Evaluation**

3.2.1 Importing the Needed Libraries



3.2.2 Reading The Data

3.2.3 Preprocessing The Data

3.2.3.1 Checking The Data

```
data.info()
```

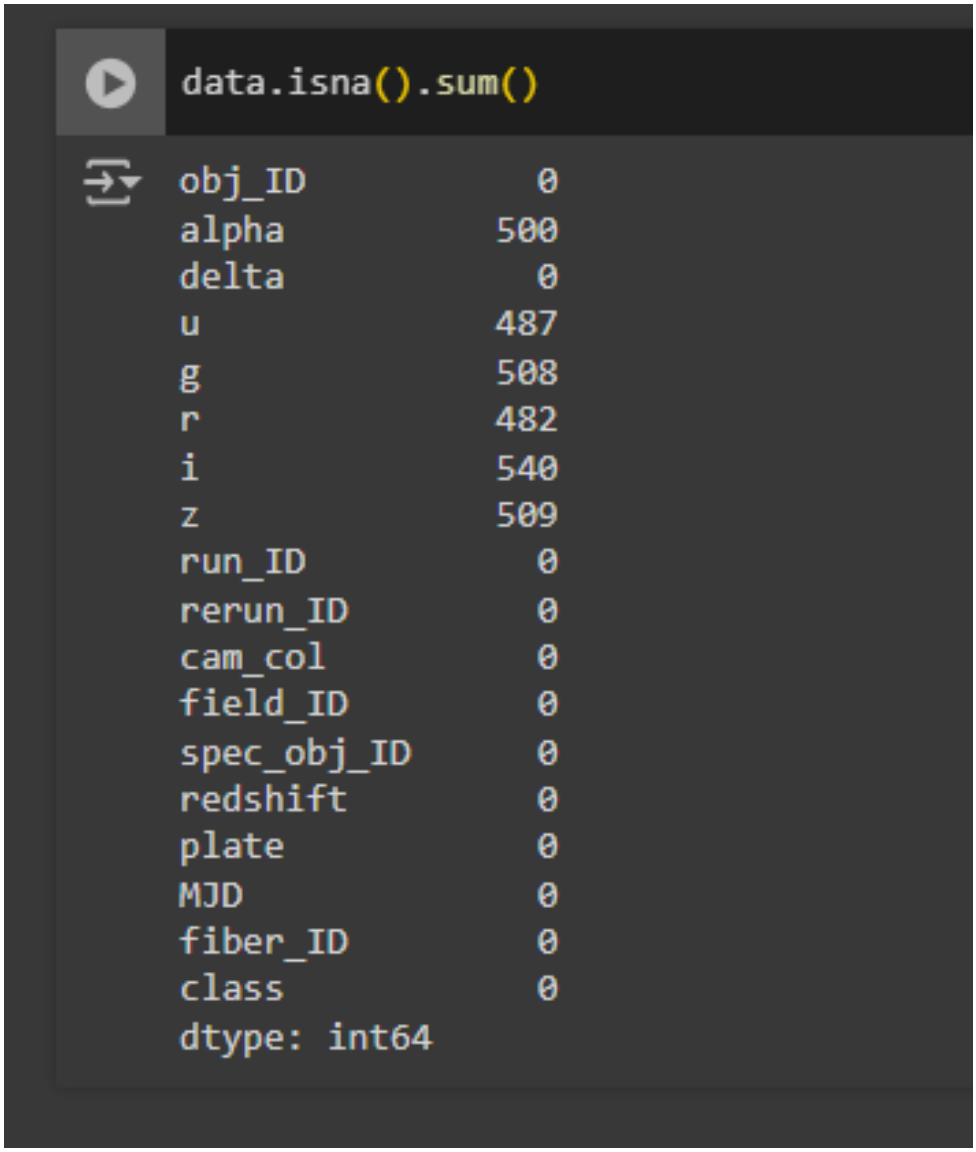
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 18 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   obj_ID       100000 non-null  float64
 1   alpha        99500 non-null   float64
 2   delta        100000 non-null  float64
 3   u            99513 non-null   float64
 4   g            99492 non-null   float64
 5   r            99518 non-null   float64
 6   i            99460 non-null   float64
 7   z            99491 non-null   float64
 8   run_ID       100000 non-null  int64  
 9   rerun_ID     100000 non-null  int64  
 10  cam_col      100000 non-null  int64  
 11  field_ID     100000 non-null  int64  
 12  spec_obj_ID  100000 non-null  float64
 13  redshift     100000 non-null  float64
 14  plate        100000 non-null  int64  
 15  MJD          100000 non-null  int64  
 16  fiber_ID     100000 non-null  int64  
 17  class        100000 non-null  object 
dtypes: float64(10), int64(7), object(1)
memory usage: 13.7+ MB
```

This line uses the data.info() method to provide a concise summary of the DataFrame data, including the column names, the count of non-null values in each column, and the data types of the columns. This information is crucial for understanding the structure and content of the dataset, allowing for effective data exploration and preprocessing.

```
data['class'].value_counts()
```

```
class
GALAXY    59445
STAR      21594
QSO       18961
Name: count, dtype: int64
```

We utilize this method to assess the distribution of values in order to determine the extent of data balancing required.



This shows the number of empty values in each column in our dataset before the preprocessing step.

3.2.3.2 Data Cleaning

```
data.info()
```

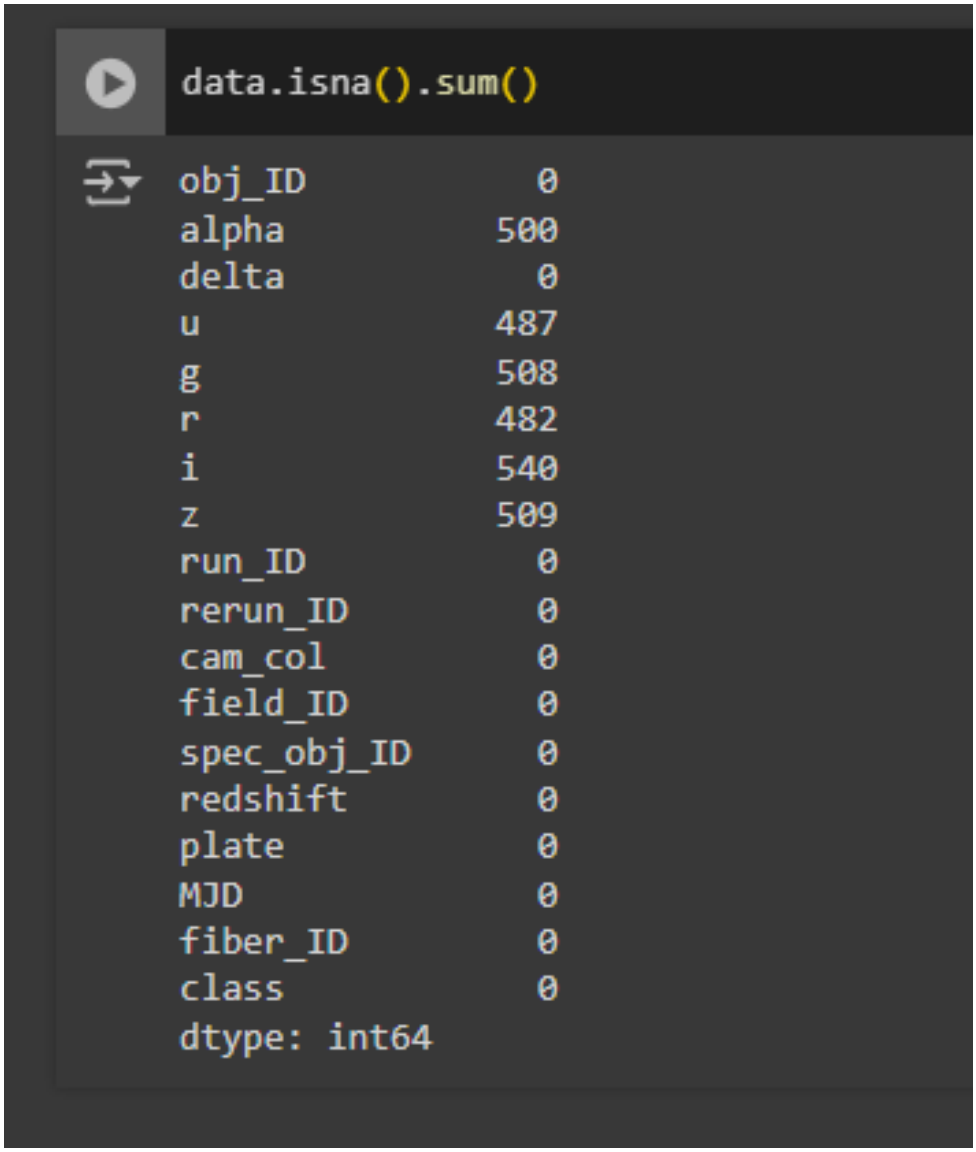
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 18 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   obj_ID       100000 non-null  float64
 1   alpha        99500 non-null   float64
 2   delta        100000 non-null  float64
 3   u            99513 non-null   float64
 4   g            99492 non-null   float64
 5   r            99518 non-null   float64
 6   i            99460 non-null   float64
 7   z            99491 non-null   float64
 8   run_ID       100000 non-null  int64  
 9   rerun_ID     100000 non-null  int64  
 10  cam_col      100000 non-null  int64  
 11  field_ID     100000 non-null  int64  
 12  spec_obj_ID  100000 non-null  float64
 13  redshift     100000 non-null  float64
 14  plate        100000 non-null  int64  
 15  MJD          100000 non-null  int64  
 16  fiber_ID     100000 non-null  int64  
 17  class        100000 non-null  object 
dtypes: float64(10), int64(7), object(1)
memory usage: 13.7+ MB
```

This line uses the data.info() method to provide a concise summary of the DataFrame data, including the column names, the count of non-null values in each column, and the data types of the columns. This information is crucial for understanding the structure and content of the dataset, allowing for effective data exploration and preprocessing.

```
data['class'].value_counts()
```

```
class
GALAXY    59445
STAR      21594
QSO       18961
Name: count, dtype: int64
```

We utilize this method to assess the distribution of values in order to determine the extent of data balancing required.



This shows the number of empty values in each column in our dataset before the preprocessing step.

```
[ ]  def handle_data_balance(data):
       num_of_deleted_rows = 37000
       # get random rows to be deleted
       deleted_rows = data[data['class'] == 'GALAXY'].sample(num_of_deleted_rows).index
       data.drop(deleted_rows,inplace = True)
```

The purpose of the "handle data balance" function is to balance a dataset by reducing the number of rows for a specific class. In our case, the function focuses on the class labeled 'GALAXY' which is our target.

```python
def handle_non_numercal_values(data):

    # convert non numercal values to integers
    non_numercal_cols = ['class']

    for column in non_numercal_cols:
        unique_values = data[column].unique()
        int_value = 1
        for value in unique_values:
            data[column] = data[column].replace({value: int_value})
            int_value+=1
        # Convert the column dataType to integer
        data[column] = data[column].astype(int)
```

The purpose of the "handle non-numerical values" function is to convert non-numerical categorical values in a dataset into numerical values, making them suitable for use in machine learning algorithms, such as K-Nearest Neighbors (KNN) and Logistic Regression, which typically require numerical input. In our case, the target column "class" contains string values.

```python
def handle_noise_values(data):
  #Replace all negative values with NAN
  data[data < 0] = np.nan
```

The purpose of the "handle noise values" function is to clean the dataset by addressing noise, specifically by replacing any negative values with NAN. This is based on the assumption that the dataset's features should not contain negative values. We choose NAN because it is the default value for empty slots, which facilitates the filling of the missing values in the next steps of data preprocessing.

```python
[ ] def handle_missing_values(data):
      #Fill other columns with the mean (double values)
      data.fillna(data.mean() , inplace=True)
```

The purpose of the "handle missing values" function is to handle missing values within the dataset by filling them with the mean value of each respective column. This includes both the originally empty values in the dataset and the noisy ones that were replaced with NAN during preprocessing.

```python
def preprocess(data):

    #Make the data balance
    handle_data_balance(data)

    #Convert all cloumns type to integer
    handle_non_numercal_values(data)

    #Handle noisy data
    handle_noise_values(data)

    #Fill the missing values
    handle_missing_values(data)


    # Remove unnecessary columns
    unnecessary_columns = ['obj_ID', 'run_ID', 'rerun_ID', 'cam_col',
                           'field_ID', 'spec_obj_ID', 'plate', 'MJD',
                           'fiber_ID']
    data.drop(columns = unnecessary_columns, inplace=True)
```

The aim of these lines of code is to call the data cleaning functions that were expressed above and remove unnecessary columns from the dataset. This step is crucial for improving computational efficiency, reducing model complexity, and mitigating the risk of overfitting.

```
[ ] preprocess(data)

data.isna().sum()

alpha       0
delta       0
u           0
g           0
r           0
i           0
z           0
redshift    0
class       0
dtype: int64

[ ] data['class'].value_counts()

class
1    22445
3    21594
2    18961
Name: count, dtype: int64
```

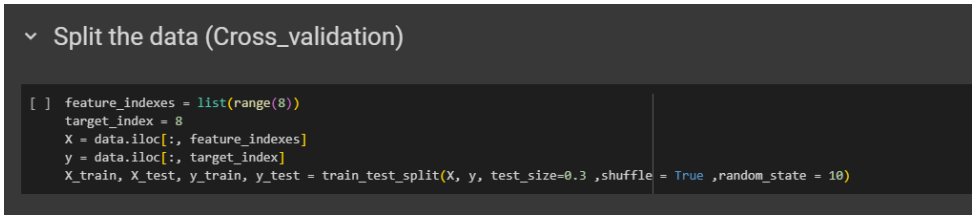
This shows the number of empty values in each column and the number of values in the "class" column (the target one) after the preprocessing step.

```
[ ] data.head()
```

	alpha	delta	u	g	r	i	z	redshift	class
1	144.826101	31.274185	24.77759	22.83188	22.584440	21.16812	21.61427	0.779136	1
3	338.741038	28.242260	22.13682	23.77656	21.611620	20.50454	19.25010	0.932346	1
5	340.995120	20.589476	23.48827	23.33776	21.321950	20.25615	19.54544	1.424659	2
6	23.234926	11.418188	21.46973	21.17624	20.928290	20.60826	20.42573	0.586455	2
7	5.433176	12.065186	22.24979	22.02172	19.676572	19.48794	18.84999	0.477009	1

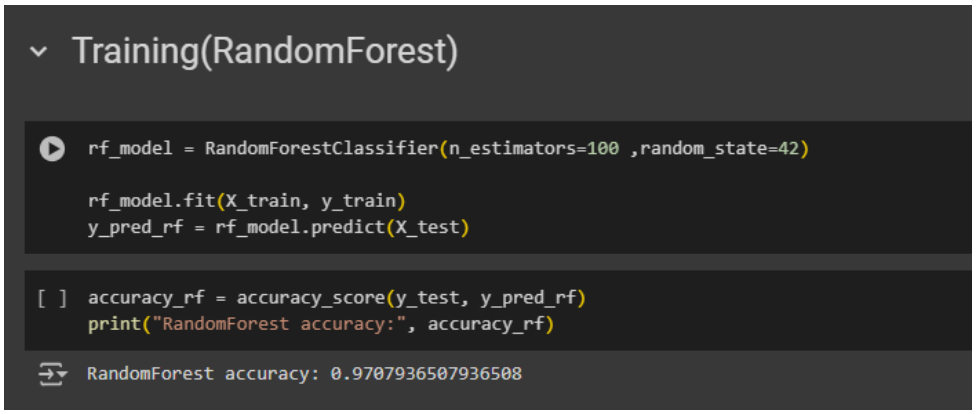
This shows the first 5 rows of our dataset after the preprocessing step.

3.2.4 Splitting The Data

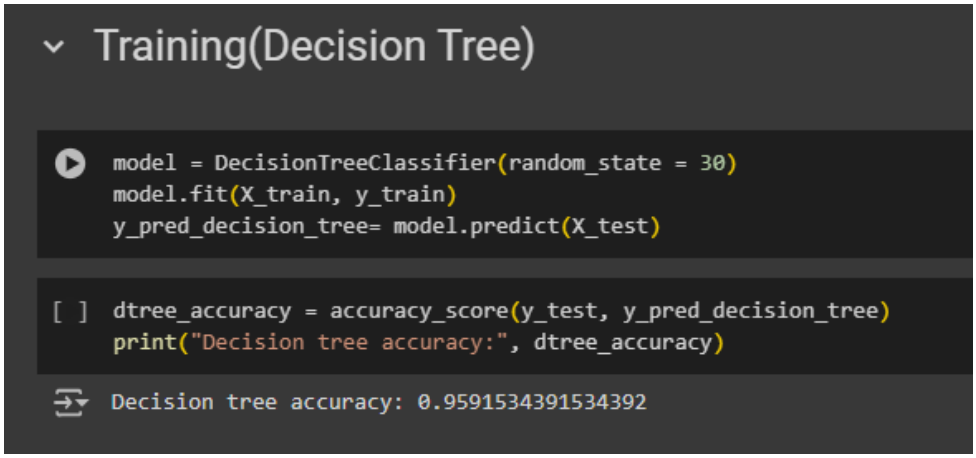


The purpose of this code is to prepare the dataset for machine learning tasks by partitioning it into features (X) and the target variable (y). Initially, it identifies the indexes of the first 8 columns as features and the 9th column as the target variable. The testing data is chosen to be 30% of the total dataset, leaving 70% for training purposes.

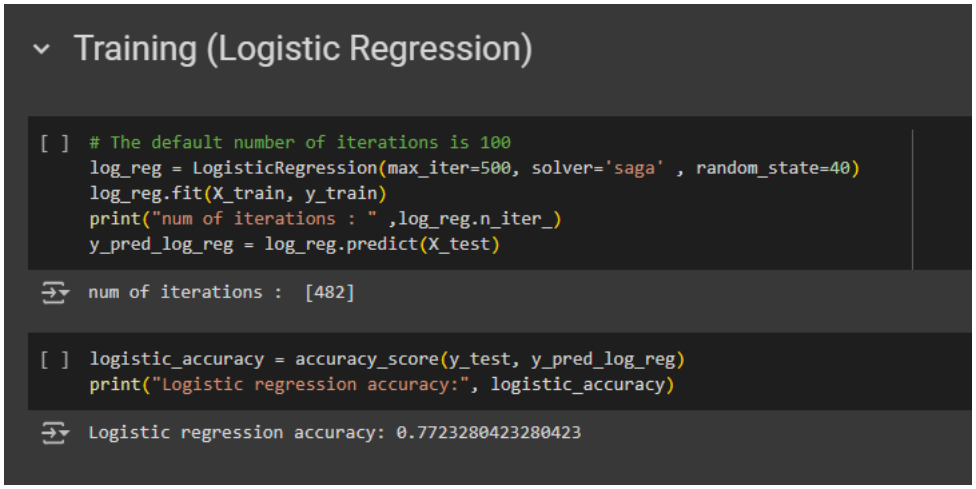
3.2.5 Model Training And Evaluation



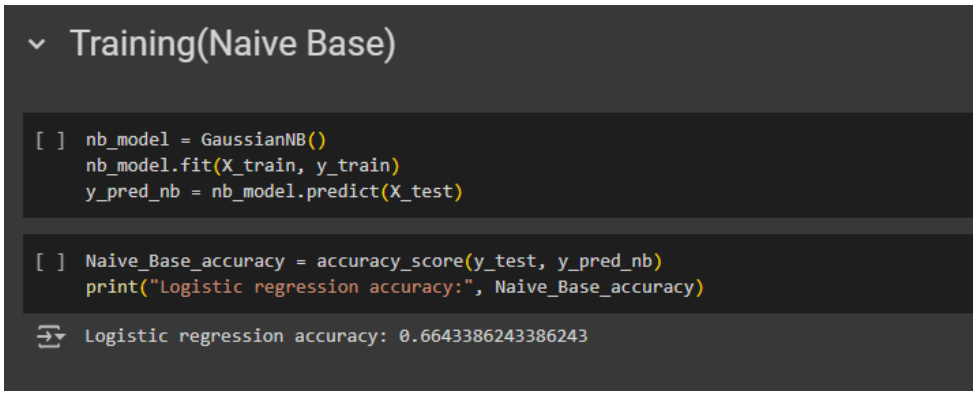
This represents our most successful experiment exploring various models for our problem. Here, we employ the Random Forest algorithm, utilizing a forest of 100 decision trees, which yields an accuracy rate of 97%.



This represents another attempt using the Decision Tree classifier, which yields an accuracy of 95%.



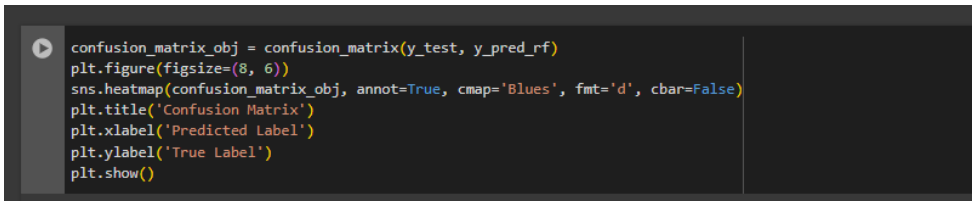
This is another attempt using the Logistic Regression classifier, which yields an accuracy of 77%.



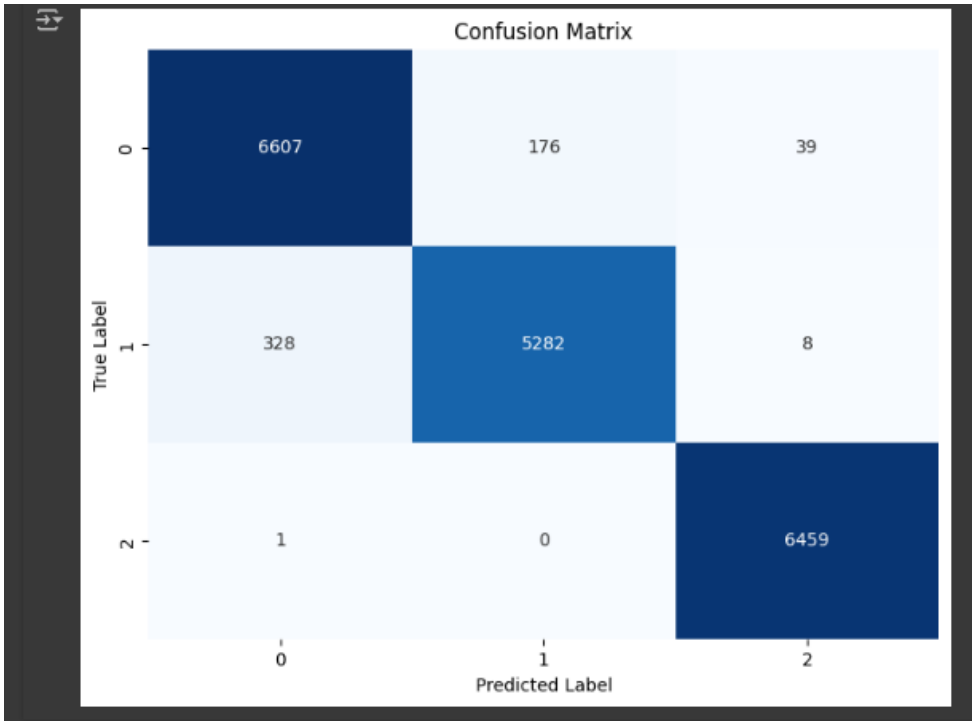
This signifies another attempt employing the Naive Bayes classifier, showcasing the lowest accuracy rate obtained at 64%.

This marks our final attempt utilizing the K-Nearest Neighbors (KNN) algorithm with a parameter value of 5 (n=5), resulting in an accuracy rate of 74%.

4 Results and Technical Discussion



creating our confusion matrix.



confusion matrix results.

4.1 Test

```
[ ]  X_new = np.array([39.149691, 28.102842,
                       21.74669, 20.03493, 19.175530,
                       18.81823, 18.65422, 0.852866]).reshape(1, -1)

     y_pred_new = rf_model.predict(X_new)

     print("Predicted class label for the new data point:", y_pred_new)

     Predicted class label for the new data point: [3]
```

This is a test for a new point.

```
] X_new = np.array([39.149691, 28.102842, 21.74669,
                    20.03493, 19.175530, 18.81823,
                    18.65422, 0.852866]).reshape(1, -1)

  y_pred_new = rf_model.predict(X_new)

  print("Predicted class label for the new data point:", y_pred_new)

  Predicted class label for the new data point: [3]
```

Test2

5 Conclusions

5.1 Conclusions

Effective Classification Achieved:

The project successfully demonstrated the capability of various machine learning models, particularly the Random Forest classifier, in classifying celestial objects into stars, galaxies, and quasars with high accuracy. This success underscores the potential of machine learning in the field of astronomy for handling large and complex datasets. Key Features Identified:

Through feature importance analysis, the project identified critical features such as redshift, magnitude in different bands (u, g, r, i, z), and other photometric properties that significantly contribute to the classification process. These features play a crucial role in distinguishing between different types of celestial objects. Data Preprocessing Significance:

The meticulous preprocessing steps, including normalization, handling missing values, and balancing the dataset, were instrumental in enhancing the model performance. These steps ensured that the machine learning algorithms could effectively learn from the dataset without being biased or misled by anomalies. Model Comparison Insights:

The comparison of different machine learning models highlighted the strengths and limitations of each. While Random Forests provided the best overall performance, the analysis showed that other models like k-NN and Logistic Regression also have their unique advantages and can be useful in specific scenarios or as part of an ensemble approach.

5.2 Recommendations for Future Work

Incorporation of Advanced Features:

Future studies should consider incorporating more advanced features, such as spectral line information, time-series data from variable stars, and higher-resolution photometric data. These additional features could help in capturing more subtle differences between celestial objects and improving classification accuracy. Exploration of Deep Learning Models:

Advanced deep learning techniques, such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), should be explored. These models are particularly effective for handling high-dimensional data and can potentially uncover more complex patterns that traditional machine learning models might miss. Transfer Learning Applications:

Utilizing transfer learning by fine-tuning pre-trained models on astronomical datasets can enhance the model's performance. Leveraging models trained on large, diverse astronomical data can provide a solid foundation and improve accuracy when adapted to specific datasets like SDSS. Development of Real-time Classification Systems:

Implementing a real-time classification system would be beneficial for ongoing astronomical surveys. Such a system could immediately classify new celestial objects as they are discovered, providing instant insights and allowing for timely follow-up observations. Automated Hyperparameter Tuning:

Future work should include automated hyperparameter tuning using methods such as Bayesian optimization, grid search, or random search. These techniques can systematically find the best hyperparameters for the models, leading to improved performance and robustness. Ensemble Methods for Improved Performance:

Employing ensemble methods, such as stacking, bagging, or boosting, can combine the strengths of multiple models to enhance overall classification performance. Ensembles can reduce overfitting and increase the robustness and accuracy of predictions. User-friendly Interfaces and Tools:

Developing user-friendly tools and interfaces for astronomers to interact with the classification models, visualize results, and provide feedback would enhance practical applicability. Such tools can bridge the gap between complex machine learning models and domain experts, making the technology more accessible. Integration with Existing Astronomical Infrastructures:

Integrating the classification system with existing astronomical databases and survey platforms can streamline the workflow and facilitate comprehensive analyses. This integration would enable seamless access to updated data and predictions, fostering a more collaborative and efficient research environment. Longitudinal Studies and Temporal Analysis:

Conducting longitudinal studies that analyze temporal changes in celestial objects could provide deeper insights into their behavior and evolution. Integrating temporal analysis with the classification models can help in identifying patterns and trends that are not apparent in static data. By addressing these recommendations, future research can build upon the foundation laid by this project, leading to more accurate, efficient, and versatile systems for celestial object classification. These advancements will

not only enhance our understanding of the universe but also provide powerful tools for the astronomical research community.

6 References

- **Dataset**: Stellar Classification Dataset
- **Random Forest Algorithm**:Stellar Classification Dataset

7 Appendix

Project Source Code: Stellar Classification.