IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2023.0322000

KNN for Stellar Classification Dataset - SDSS17

KHADKA PILOT[1]* AND POUDEL KSHITIZ[2]*

[1]Institute of Engineering, Thapathali Campus, Thapathali, Nepal (e-mail: pilot.076bct025@tcioe.edu.np)
[2]Institute of Engineering, Thapathali Campus, Thapathali, Nepal (e-mail: kshitizpoudel18@gmail.com)

Corresponding author: Khadka Pilot (e-mail: pilot.076bct025@tcioe.edu.np).

* Authors contributed equally

ABSTRACT Stellar classification, a fundamental task in astrophysics, involves categorizing celestial objects based on their intrinsic properties. In this study, we explore the application of the k-nearest neighbors (KNN) algorithm to the classification of stars. Leveraging a dataset containing attributes such as spectral features, magnitudes, and redshifts, we employ the KNN algorithm to determine the class of stars, considering factors like proximity and shared characteristics. Through experimentation and analysis, we investigate the impact of various KNN parameters and distance metrics on classification accuracy. Our results showcase the effectiveness of KNN in discerning star classes, providing insights into the suitability of this approach for stellar classification within the context of modern astronomy. This work contributes to the advancement of automated stellar classification methodologies and their potential role in large-scale sky surveys and data-driven astrophysical research.

INDEX TERMS KNN, Stellar classification

I. INTRODUCTION

STELLAR classification, a cornerstone of modern astronomy, is instrumental in understanding the diversity and evolution of celestial objects within the universe. The categorization of stars based on their intrinsic characteristics provides essential insights into their origins, lifecycles, and broader astrophysical phenomena. Traditional methods of stellar classification, relying on human expertise and visual analysis, have been limited in their scalability and subjectivity. With the advent of advanced data acquisition techniques and the proliferation of large astronomical datasets, the need for automated and data-driven approaches to star classification has become increasingly evident.

In this context, machine learning algorithms have emerged as powerful tools to tackle complex astronomical tasks. Among these, the k-nearest neighbors (KNN) algorithm stands out as a versatile and intuitive approach for classification tasks. KNN, rooted in the principle of proximity-based classification, determines the class of an unknown object by considering the classes of its nearest neighbors. Applied to the domain of stellar classification, the KNN algorithm has the potential to leverage diverse attributes such as spectral features, magnitudes, and redshifts to accurately categorize stars across various classes. In this study, we delve into the application of the KNN algorithm to the intricate task of stellar classification. Our objective is to explore the efficacy of this approach in discerning star classes and understanding its adaptability to the complex and nuanced features that characterize stars. Through a comprehensive analysis, we investigate the interplay between KNN parameters, distance metrics, and feature selection techniques, aiming to optimize the algorithm's performance for stellar classification.

II. METHODOLOGY

A. THEORY

K-nearest neighbors (KNN) is a supervised machine learning algorithm that identifies a group of k objects in the training set that are close to the test object and assigns a label based on the predominance of a particular class within this neighborhood.

Given a training set D and a test object z, represented as a vector of attribute values with an unknown class label, the algorithm computes the distance (or similarity) between z and all training objects to determine its nearest neighbor list. It then assigns a class to z by considering the majority class of neighboring objects.

Commonly, the Euclidean distance or Manhattan distance is used to calculate the distance between two points x and y with n attributes. For Euclidean distance, the formula is:

$$\text{Euclidean distance}(x_i, x_j) = \sqrt{\sum_{s=1}^{p} (x_i s - y_j s)^2} \qquad (1)$$

For Manhattan distance, the formula is:

$$\text{Manhattan distance}(x_i, x_j) = \sum_{s=1}^{p} |x_i s - y_j s| \tag{2}$$

In general, both measures can be seen as special cases of the Minkowski distance:

$$d(x_i, x_j) = \left(\sum_{s=1}^{p} |x_{is} - x_{js}|^q \right)^{\frac{1}{q}} \tag{3}$$

Hamming distance is used when dealing with categorical attributes. It quantifies the dissimilarity between two attribute vectors x and y of equal length:

$$\text{Hamming distance}(x_i, x_j) = \frac{1}{n} \sum_{s=1}^{p} \delta(x_{is}, y_{js}) \tag{4}$$

Where $\delta(x_{is}, y_{js})$ is the Kronecker delta function, equal to 1 when $x_{is} \neq y_{js}$ and 0 otherwise.

Certain distance metrics can be influenced by the high-dimensional nature of the data. For instance, the Euclidean distance becomes less discriminating as the number of attributes grows. To prevent the potential dominance of a single attribute over distance calculations, it might be necessary to scale the data.

Let k_r denote the count of observations from the group of nearest neighbors that belong to class r:

$$\sum_{r=1}^{c} k_r = k \tag{5}$$

Then, the new observation is predicted to be in class l with

$$k_l = \max_r(k_r) \tag{6}$$

The consideration of k-neighbours prevents the observation to be influenced by a single outlier or class. The locality of this technique is determined by the parameter k: With $k = 1$, we have a single nearest neighbour with maximal locality, while as $k \to n_L$, a majority vote is taken. Consequently, this implies a consistent prediction for all new observations requiring classification: The most frequent class within the learning set is predicted.

B. SELECTING K

When predicting with k nearest neighbors, the prediction is always an average over k data points. Increasing k means averaging more data, which is beneficial as it reduces noise and improves precision.

On the other hand, decreasing k leads to averaging over fewer data points, allowing to focus on points close to prediction location. However, this approach can be sensitive to noise points in the data, potentially leading to less reliable predictions.

There exists a trade-off between bias and variance, approximation versus estimation error, and accuracy versus precision when selecting k.

One method to find an optimal value for k is through cross-validation. By repeatedly splitting the data into different training and testing sets, and averaging the performance across these splits, we can better assess the predictive capabilities of the model with different values of k and choose the most suitable k for the problem at hand.

$$\frac{v-1}{v} \tag{7}$$

Because many random splits are made, noise is reduced in the estimate compared to simple data splitting. On the other hand, each training set contains a fraction of the complete data, so what generalizes best at a sample size close to that of the full data is observed.

$$k = \sqrt{n} \tag{8}$$

Another method is to use the square root of n as starting point for selecting the k value.his heuristic is based on the idea that a smaller k value can help capture local patterns in the data, while a larger k value can lead to a smoother decision boundary. However, the optimal value of k can vary based on characterisitcs of the data.

C. CURSE OF DIMENSIONALITY

The kNN classifier makes an assumption that data points that are near each other share similar labels. However, in high-dimensional spaces, points drawn from a probability distribution tend to be far apart.The range of distances between data points decreases, and the distribution of distances becomes more compact.

To understand this concept better, a simplified example in 2D can be considered and then extended to higher dimensions:

Imagine a unit square with sides of length 1. When we randomly scatter points within this square, the distances between points can vary widely. Some points might be very close to each other, while others are far apart. This results in a relatively spread-out distribution of distances.

As we move into higher dimensions (3D, 4D, and beyond), the concept of "volume" in the space undergoes a remarkable change. With each additional dimension, the "volume" of the space increases exponentially. This expansion has notable implications for how data points are distributed.

In a 2D space, such as a square, the area grows linearly as you increase the side length. In 3D (like a cube), the volume grows cubically with the side length, and this trend persists as dimensions increase. The available space fans out in a way that creates more room for data points to inhabit.

However, an intriguing phenomenon occurs as dimensions increase. Although the space itself grows exponentially, the majority of this "volume" becomes concentrated near the edges or corners of the higher-dimensional hypercube. As a result, the "neighborhood" of any given point becomes increasingly concentrated around the corners or edges. Points within this neighborhood will be relatively close to each other in terms of distance.

This is why distances become more similar in higher dimensions. The points are distributed in a way that the majority of distances are relatively small compared to the overall size of the space. In other words, the distances between points are "compressed" in higher dimensions, leading to a concentration of distances within a small range. This effect is often referred to as the "curse of dimensionality," and the concept of distance becomes less reliable in high-dimensional spaces.

This can be illustrated by generating random points from uniform distribution on a d-cube and computing their Euclidean distance.

Illustrated in the figure(), 1000 points were randomly generated in separate dimensions. When the distance distribution was plotted, a Gaussian distribution was observed with the average relative distance between the data points increasing as the dimension increases.

To counteract the effects of dimensionality and calculate the relative distance between points in a higher-dimensional space, one approach is to normalize or scale the distance measures based on the dimensionality. The normalized distance is given by:

$$\text{normalized\_distance} = \frac{\text{euclidean\_distance}}{\sqrt{\text{dimension}}} \tag{9}$$

where:

- euclidean_distance is the standard Euclidean distance between the two points in the high-dimensional space.
- dimension is the number of dimensions in the space.

This normalization ensures that the calculated distance is adjusted based on the dimensionality, making it more meaningful and comparable across different dimensions.

Figure() demonstrates the curse of dimensionality. The pariwise distance between data points for higher diemsion is shown in the histogram plot. As the number of dimensions d grows, all distances concentrate within a very small range.

Increasing the number of training samples seems to be first assumption in mitigating such effects, and it may be true for smaller dimensions. But the needed data points grows exponentially.

Algorithms like k-nearest neighbors, which rely on measuring distances to determine proximity, can become less effective because the concept of "closeness" becomes less informative when distances become similar and neighborhood boundaries become less well-defined. This can lead to challenges in making accurate predictions or classifications based on distance-based methods in high-dimensional spaces.

D. WEIGHTED KNNS

The concept of weighted KNN is based upon the concept that instances in the training dataset which exhibit close distance to the new observation (y, x) should be assigned a higher weight than instances that are far away from the data point.

In the traditional k-Nearest Neighbors (kNN) approach, only the k nearest neighbors influence the prediction. However, this influence remains uniform across all these neighbors, even when their respective similarities to (y, x) significantly vary. To address this challenge, the distance metrics used during the initial neighbor search can serve as weights.

A popular weighting scheme is using the inverse squared distance:

$$w[i] = \frac{1}{d(x[i], x[t])^2} \tag{10}$$

where $h(x) = f(x)$ is used for an exact match. Other strategies include adding a small constant to the denominator to avoid zero-division errors.

The weighted Euclidean distance for the weighted k-nearest neighbors (KNN) can be represented as:

$$\text{Weighted Euclidean distance}(x_i, x_j) = \sqrt{\sum_{s=1}^{p} w_s (x_{is} - y_{js})^2} \tag{11}$$

where w_i is the weight assigned to the data point x_i, and p is the number of dimensions in the feature space.

1) Data with low dimensional structure

Data may lie in a lower dimensional space. For example, consider images like handwritten digits or faces. In these cases, the dimensionality of the data can be significantly lower than its space. Despite the images of faces taking up megabytes, lower dimensional representation may require fewer than 50 attributes (e.g., hair color, eye color) that capture the key variations among faces.

Techniques like PCA and kernel methods can be useful when the actual dimensionality of the dataset is much lower than its representation. These methods enable effective dimensionality reduction while retaining essential information. Another technique for data reduction involves representing data with fewer observations.

2) Approximate KNNs

In cases where finding approximate solution is acceptable, Approximate KNNs can be used. These methods allows for solution where the cost of computation is too high. Few commonly used approximate KNN techniques are: Locality sensitive hashing: It uses hash function to map similar data points to same probability buckets. Random projection: It involves mapping high dimensional structure to lower dimensional space while preserving the pairwise distance between points.While it may introduce some level of approximation, it often retains the essential geometric relationships in the data.

III. COMPUTATIONAL ASPECTS

A. MEMORY COMPUTATION

To implement KNN, we need to keep the entire dataset, which consists of n data points, each with p features and 1 label. Therefore, the total memory cost is $O(n(p+1)) = O(np)$.

TIME COMPLEXITY

When making a prediction for a new point, we need to compute the distance between the new point and every data point

in the dataset, where each point has p features. Computing the distance for one point takes $O(p)$ time. For all n data points, it takes $O(np)$ time to compute all the distances.

After computing all the distances, we need to find the k smallest distances out of the n distances, which can be done in $O(n)$ time. Once we have the k nearest neighbors, averaging their labels takes $O(k)$ time.

Therefore, the total time complexity for KNN is $O(np + n + k) = O(np + k)$.

When the dataset is large (e.g., $n = 10^5$), both time and space complexity become significant. To handle large datasets and reduce computation time, we can consider using fewer than n data points. One approach is to pre-select possible neighbors using data structures like the k-d tree, which efficiently organizes the data points in a way that allows for faster search for nearest neighbors.

B. ALGORITHM

Algorithm 1 k-Nearest Neighbors (k-NN) Algorithm

function kNN($\mathcal{D}$, x_{new}, k)
  Input: $\mathcal{D}$ = Training dataset $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
    x_{new} = new data point
    k = Number of neighbors
  Output: Predicted label x_{new}
  for $i \leftarrow 1$ **to** n **do**
    Calculate distance $d(x_{\text{new}}, x_i)$
  end for
  Sort (d)
  Select top k distances
  Perform majority voting among the k neighbors
  Return Predicted label or value for x_{new}
end function

Algorithm 2 V-Fold Cross-Validation

function vFoldCrossValidation(data, v)
  Inputs:
  D = Dataset
  v = Number of folds

  fold_size $\leftarrow \frac{\text{size(data)}}{v}$
  validation_scores $\leftarrow$ []
  for $i = 1$ **to** v **do**
    val_start $\leftarrow (i-1) \times$ fold_size
    val_end $\leftarrow i \times$ fold_size
    train_set $\leftarrow$ data[exclude val_set]
    model $\leftarrow$ train_model(train_set)
    val_score $\leftarrow$ validate_model(model, val_set)
    val_scores.append(validation_score)
  end for
  average_score $\leftarrow \frac{\sum \text{validation\_scores}}{k}$
  Return average_score
end function

C. SYSTEM BLOCK DIAGRAM

Appendix

D. METHODOLOGY AND TOOLS

This project involved use of various tools and libraries that played a crucial role in the analysis of the stellar classification dataset. The main tools and methods used are as follows:

- `pd.read_csv`: Used to import the stellar dataset stored in a CSV file for structured data input.
- `df.drop`: Employed to remove less significant attributes like object ID, field ID, and fiber ID, streamlining the dataset.
- `plt.histplot`: Utilized for creating histograms that display the distribution of instances across different classes (galaxies, quasars, stars).
- `plt.boxplot`: Used to identify outliers in the data by generating boxplots, revealing extreme values.
- `sns.kdeplot`: Enabled the creation of kernel density plots, visually representing density distributions of redshift values.
- `sklearn.preprocessing.LabelEncoder`: Applied for encoding classes into numerical values for processing.
- `sklearn.preprocessing.StandardScaler`: Used to standardize attribute scales, enhancing model training.
- `sklearn.neighbors.KNeighborsClassifier`: Implemented K-nearest neighbors (KNN) algorithm for predicting classes based on data point proximity.
- `sklearn.metrics.classification_report`: Employed to generate classification reports, providing insights into model performance across classes.

These tools collectively supported data handling, visualization, preprocessing, model implementation, and evaluation stages. The methods contributed to the development of a reliable stellar classification model, shedding light on the distinct attributes of galaxies, quasars, and stars.

IV. WORKING PRINCIPLE

A. DATASET DESCRIPTION

The dataset encompasses the following attributes:

- **obj_ID**: Object ID, a distinct identifier for each celestial entity present in the dataset.
- **alpha**: Right Ascension (RA), indicating the angular distance of a celestial object eastward along the celestial equator from the vernal equinox.
- **delta**: Declination (Dec), denoting the angular distance of a celestial object towards the northern or southern region of the celestial equator.
- **u**: Apparent brightness magnitude in the ultraviolet (u) wavelength band.
- **g**: Apparent brightness magnitude in the green (g) wavelength band.
- **r**: Apparent brightness magnitude in the red (r) wavelength band.

- **i**: Apparent brightness magnitude in the infrared (i) wavelength band.
- **z**: Apparent brightness magnitude in the near-infrared (z) wavelength band.
- **run_ID**: Identification number linked to a specific observational run.
- **rerun_ID**: Identification number associated with a particular rerun of observational data.
- **cam_col**: Identifier for the camera column used during observation.
- **field_ID**: Identifier for the observed region of the sky, often used in systematic astronomical surveys.
- **spec_obj_ID**: Spectroscopic Object ID, a unique marker for objects with spectroscopic data.
- **redshift**: Measurement of how much an object's light has shifted towards longer wavelengths due to the universe's expansion.
- **plate**: Identification number linked to the spectroscopic plate used in data collection.
- **MJD**: Modified Julian Date, signifying the date of the observation.
- **fiber_ID**: Identifier for the fiber used to gather data related to an object's spectrum.
- **class**: The class or category to which a celestial entity is categorized.

B. DATA ANALYSIS

Figure **??** demonstrates a noteworthy observation regarding the redshift values across different celestial objects. Stars exhibit lower redshift values in comparison to galaxies and QSOs. This disparity can be attributed to stars being situated within our own galaxy, resulting in slower relative motion compared to more distant celestial objects.

The logarithmic scale of redshift values on the x-axis, accompanied by the utilization of kernel density estimation, effectively illustrates the probability density distribution. This representation underscores the substantial difference in redshift values between stars and more distant objects like galaxies and QSOs. Furthermore, the visibly distinct and non-overlapping distributions suggest that stars occupy a clearly discernible space within the context of nearest neighbor algorithms.

Interestingly, unlike the redshift attribute, the distribution of other attributes appears relatively consistent across all classes. This uniformity across classes reaffirms the pivotal role of redshift as a discriminating attribute for classifying celestial objects. The weighting of distances along this dimension is anticipated to wield considerable influence in accurately distinguishing between different classes.

V. RESULT AND ANALYSIS

The original dataset comprises 100,000 instances. When training a k-nearest neighbors (KNN) model, the process involves storing the data points within a d-dimensional space. In the case of having 6 attributes, these data points reside within a six-dimensional space.

The essence of KNN's computation becomes prominent during the testing phase. At this stage, given a specific data point, the algorithm calculates the distances between that point and every point within the training data. Subsequently, the neighbors are arranged based on the minimum distances. Depending on the chosen number of neighbors, a majority voting mechanism determines the class of the given unlabeled data instance.

In terms of computational complexity, the brute force KNN approach is characterized by a space and time complexity of O(n*d), where 'n' represents the number of data points, and 'd' signifies the number of features that define each data point.

The graph provided illustrates that the optimal number of neighbors is 5. Opting for a smaller number of neighbors makes predictions increasingly susceptible to noise and outliers.

Classification Report of weighted KNN

	precision	recall	f1-score	support
0	0.97	0.97	0.97	11889
1	0.96	0.93	0.95	3792
2	0.94	0.97	0.95	4319
accuracy			0.96	20000
macro avg	0.96	0.95	0.95	20000
weighted avg	0.96	0.96	0.96	20000

The overall accuracy of 0.96 means that the model correctly classifies approximately 96% of instances across all classes. The macro average F1-score of 0.95 indicates a balanced model performance considering all classes equally. The weighted average F1-score of 0.96 gives a more realistic assessment of overall performance, accounting for class imbalances.

Classification Report of weighted KNN

	precision	recall	f1-score	support
0	0.97	0.97	0.97	11889
1	0.96	0.93	0.95	3792
2	0.95	0.99	0.97	4319
accuracy			0.97	20000
macro avg	0.96	0.96	0.96	20000
weighted avg	0.97	0.97	0.97	20000

The macro average F1-score of 0.96 emphasizes a balanced model performance across all classes, while the weighted average F1-score of 0.97 underscores the model's proficiency, considering class imbalances.

VI. CONCLUSION

Our exploration of stellar classification using the K-nearest neighbors (KNN) method has provided meaningful insights into its applicability for categorizing celestial objects. The classification report outcomes highlight the model's ability to distinguish between different classes – galaxies, quasars, and stars – based on their distinct attributes.

The weighted KNN approach has demonstrated its effectiveness in addressing class imbalances, resulting in improved accuracy and robustness across classifications. Notably, the model's performance in recognizing stars stands out due to high precision and recall values.

Although this study has yielded promising results, there is potential for further investigation. Exploring different parameters and incorporating more comprehensive features could enhance the model's performance. Additionally, the findings reinforce the value of data-driven techniques in contributing to our understanding of astronomical phenomena.

In conclusion, the KNN method holds promise as a tool for accurate stellar classification, aligning with the broader trend of leveraging machine learning to unravel the mysteries of the universe.

REFERENCES

[1] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *"Deep Learning."* MIT Press.
[2] Daniel Jurafsky & James H. Martin. *Speech and Language Processing*.
[3] M. Narasimha Murty, V. Susheela Devi. *Pattern Recognition: An Algorithmic Approach*.

PILOT KHADKA is a student at Institute of Engineering, Thapathali Campus. He is expected to graduate in Bachelor of Computer Engineering in 2024. During his time at Thapathali Campus, Pilot has actively engaged in various academic and extracurricular activities. He has participated in coding competitions, collaborated on software development projects, and demonstrated a keen interest in exploring new technologies.

KSHITIZ POUDEL is a student at Institute of Engineering, Thapathali Campus. He is expected to graduate in Bachelor of Computer Engineering in 2024. His relentless pursuit of knowledge and dedication to uplifting and empowering others make him an exceptional contributor and an invaluable asset to the academic community.

VII. APPENDIX

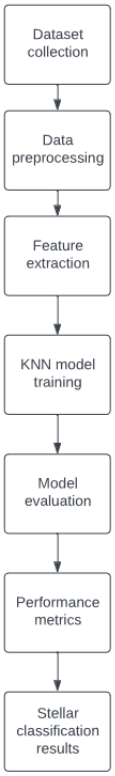

FIGURE 1. System Block Diagram

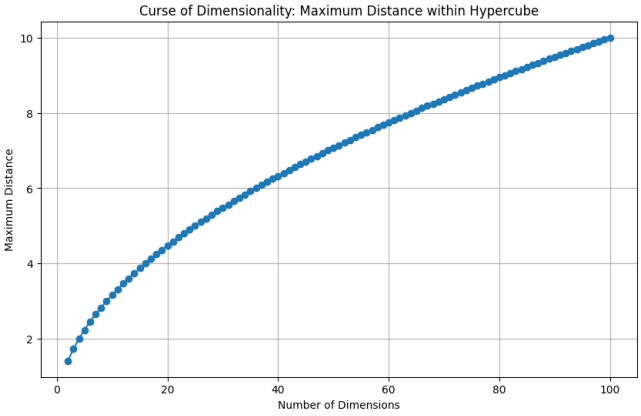

FIGURE 2. Distance in higher dimension

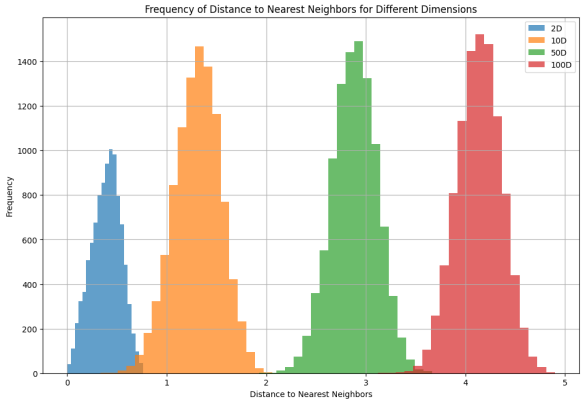

FIGURE 3. distance distribution in higher dim

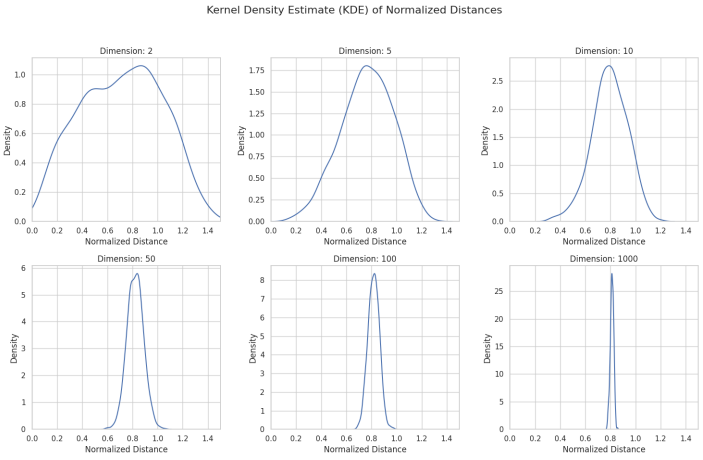

FIGURE 4. Normalized distance

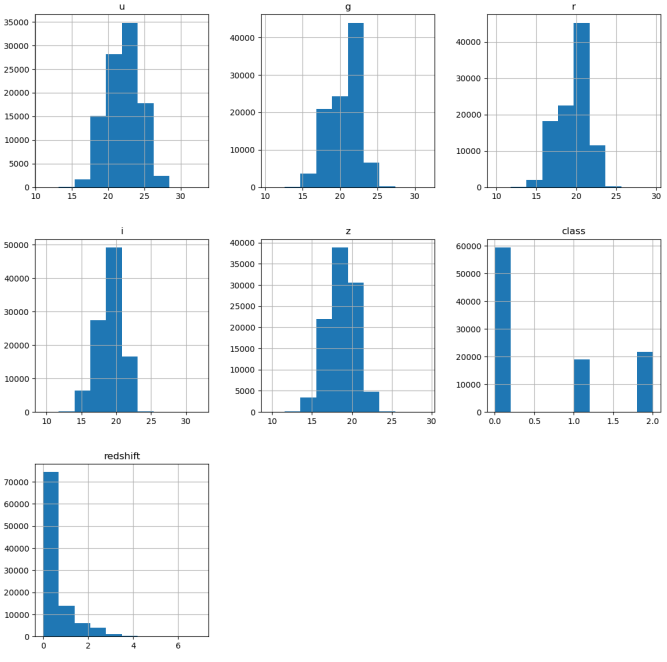

FIGURE 5. Stellar dataset distribution

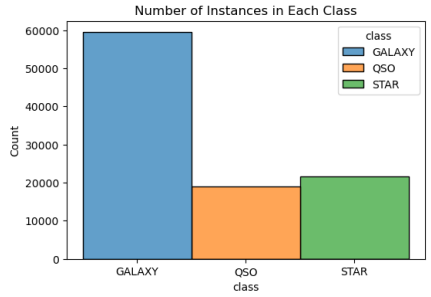

FIGURE 6. Histogram of classes

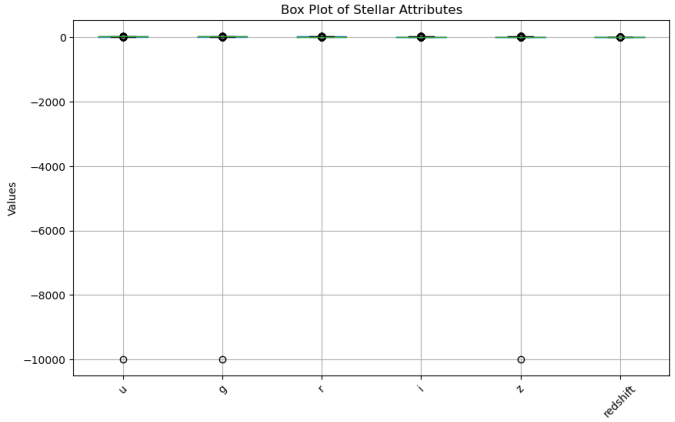

FIGURE 7. Boxplot of stellar attributes

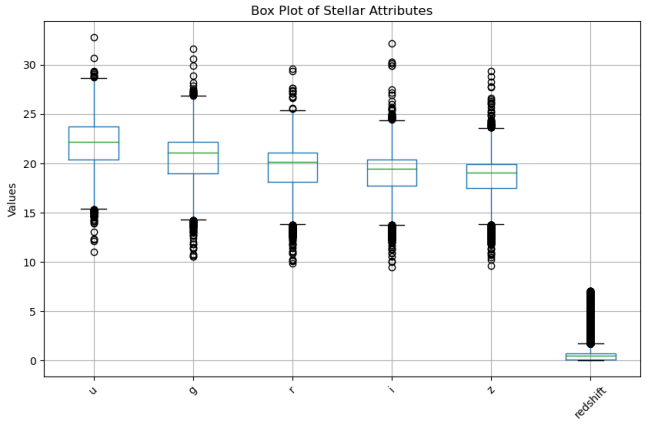

FIGURE 8. Box plot of stellar attributes, Normalized

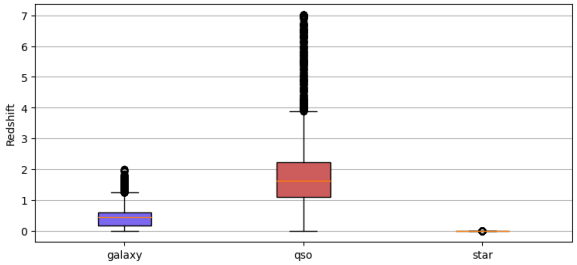

FIGURE 9. Box plot of redshift

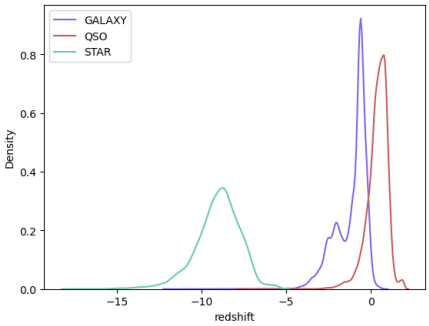

FIGURE 10. KDE plot of redshift

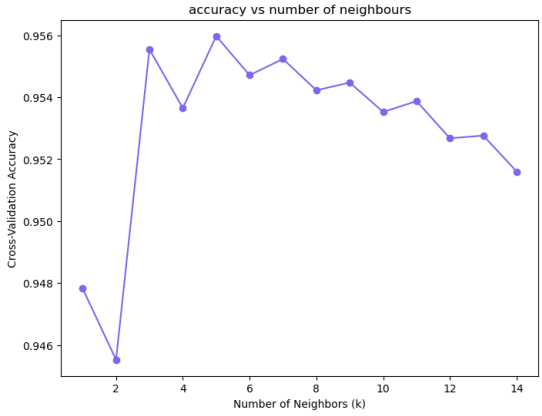

FIGURE 11. Accuracy vs number of nighbours

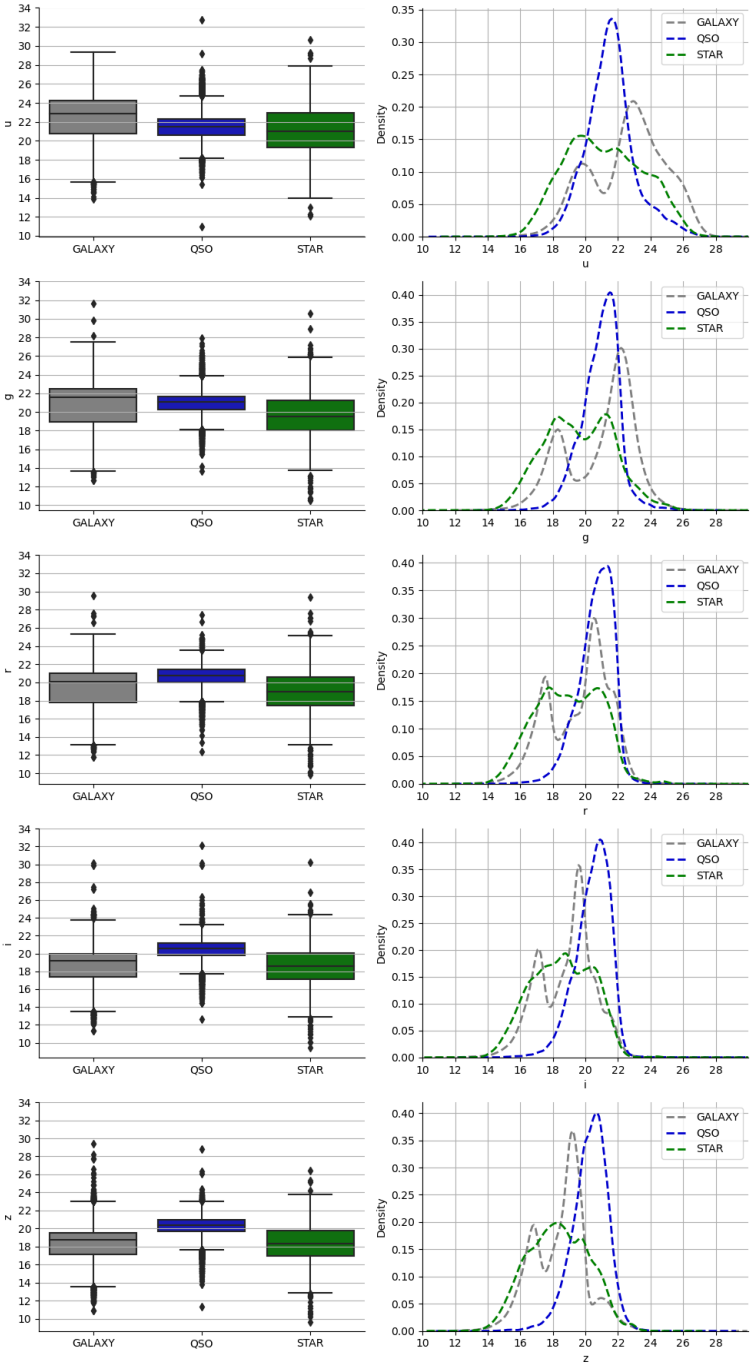

FIGURE 12. Boxplot and KDE plot

VIII. CODE

A. HIGHER DIMENSION DISTANCE CODE

```

import numpy as np
import matplotlib.pyplot as plt

# Function to generate random data points
def generate_data(num_points, num_dimensions):
    return np.random.rand(num_points,
    num_dimensions)

# Function to calculate Euclidean distance between
     two points
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))

# Number of data points
num_points = 10000

# Dimensions to plot
dimensions_to_plot = [2, 10, 50, 100]

# Number of nearest neighbors
k = 5

# Plotting histograms
plt.figure(figsize=(12, 8))

for num_dimensions in dimensions_to_plot:
    data = generate_data(num_points,
    num_dimensions)
    query_point = generate_data(1, num_dimensions)
      # Generate a single query point

    distances_to_query = [euclidean_distance(
    query_point, point) for point in data]

    plt.hist(distances_to_query, bins=20, label=f'
    {num_dimensions}D',alpha=0.7)

plt.xlabel('Distance to Nearest Neighbors')
plt.ylabel('Frequency')
plt.title('Frequency of Distance to Nearest
    Neighbors for Different Dimensions')
plt.legend()
plt.grid(True)
plt.show()

import numpy as np
import matplotlib.pyplot as plt

# Function to generate random data points within a
     unit square
def generate_data(num_points, num_dimensions):
    return np.random.rand(num_points,
    num_dimensions)

# Function to calculate maximum Euclidean distance
     within a hypercube
def max_distance_in_hypercube(num_dimensions):
    corner1 = np.zeros(num_dimensions)
    corner2 = np.ones(num_dimensions)
    return np.linalg.norm(corner2 - corner1)

# Number of data points
num_points = 100

# Range of dimensions
min_dimensions = 2
max_dimensions = 100

# Generate data points and calculate distances
dimensions = range(min_dimensions, max_dimensions
    + 1)
max_distances = []

for num_dimensions in dimensions:
    max_distances.append(max_distance_in_hypercube
    (num_dimensions))

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(dimensions, max_distances, marker='o')
plt.xlabel('Number of Dimensions')
plt.ylabel('Maximum Distance')
plt.title('Curse of Dimensionality: Maximum
    Distance within Hypercube')
plt.grid(True)
plt.show()

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Function to generate random data points
def generate_data(num_points, num_dimensions):
    return np.random.rand(num_points,
    num_dimensions)

# Function to calculate Euclidean distance between
     two points, normalized by sqrt(d)/2
def normalized_distance(point1, point2):
    dimension = len(point1)
    normalization_factor = np.sqrt(dimension) / 2
    return np.sqrt(np.sum((point1 - point2) ** 2))
     / normalization_factor

# Number of data points
num_points = 1000

# Dimensions to plot
dimensions_to_plot = [2, 5, 10, 50, 100, 1000]

# Set style for seaborn
sns.set(style="whitegrid")

# Set up subplots
num_plots = len(dimensions_to_plot)
num_cols = 3
num_rows = (num_plots + num_cols - 1) // num_cols

# Create subplots
fig, axes = plt.subplots(num_rows, num_cols,
    figsize=(15, 5 * num_rows))
fig.suptitle('Kernel Density Estimate (KDE) of
    Normalized Distances', fontsize=16)

# Generate and plot data for each dimension
for i, num_dimensions in enumerate(
    dimensions_to_plot):
    row = i // num_cols
    col = i % num_cols

    data = generate_data(num_points,
    num_dimensions)
    query_point = generate_data(1, num_dimensions)
    [0]

    normalized_distances_to_query = [
    normalized_distance(query_point, point) for
    point in data]
    axes[row,col].set_xlim(0,1.5)
    sns.kdeplot(normalized_distances_to_query, ax=
    axes[row, col])
    axes[row, col].set_title(f'Dimension: {
    num_dimensions}')
    axes[row, col].set_xlabel('Normalized Distance
```

```
    ')
    axes[row, col].set_ylabel('Density')
    axes[row, col].grid(True)

# Adjust layout
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

B. KNN CODE

```
# -*- coding: utf-8 -*-
"""Data_mining_lab4.2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ia-jx05ELNAVRZM02kwNOjgioY4FQxi6
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn

df=pd.read_csv('star_classification.csv')

df.head()

df = df.drop(['obj_ID', 'alpha', 'delta', 'run_ID', 'rerun_ID', 'cam_col', 'field_ID', 'fiber_ID', 'MJD', 'plate'], axis = 1) # drop metadata# drop metadata

df.shape

df.isnull().sum()

df.dtypes

import seaborn as sns

plt.figure(figsize=(6, 4))
sns.histplot(data=df, x='class', hue='class', alpha=.7)
plt.title('Number of Instances in Each Class')
plt.show()

df2=pd.read_csv('star_classification.csv')
df2 =df2.drop(['obj_ID', 'alpha', 'delta', 'run_ID', 'rerun_ID', 'cam_col', 'field_ID', 'fiber_ID', 'spec_obj_ID', 'MJD', 'plate'], axis = 1)
df2_no_anomaly = df2[df2['u'] > -1]

from sklearn.preprocessing import LabelEncoder, StandardScaler
LE = LabelEncoder()
df['class'] = LE.fit_transform(df['class'])

df.head()

df_no_anomaly=df[df['u'] > -1]

df_no_anomaly.head()

# Swap the last two columns
last_column = df_no_anomaly.columns[-1]
second_last_column = df_no_anomaly.columns[-2]

df_no_anomaly[last_column], df_no_anomaly[second_last_column] = df_no_anomaly[second_last_column], df_no_anomaly[last_column]

df_no_anomaly.rename(columns={'redshift': 'class', 'class': 'redshift'}, inplace=True)

df_no_anomaly.head()

df2_no_anomaly.boxplot(figsize=(10, 6))
plt.title('Box Plot of Stellar Attributes')
plt.ylabel('Values')
plt.xticks(rotation=45)  # Rotates x-axis labels for better visibility
plt.show()

df2_no_anomaly = df2[df2['u'] > -1] # remove extremely bright point

#remove anomaly and replot the box plot
df2_no_anomaly.boxplot(figsize=(10, 6))
plt.title('Box Plot of Stellar Attributes')
plt.ylabel('Values')
plt.xticks(rotation=45)  # Rotates x-axis labels for better visibility
plt.show()

df2_no_anomaly.hist(bins = 10 , figsize= (14,14))
plt.show()

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder

# Your data loading and preprocessing steps

# Split data by class
galaxy = df2_no_anomaly[df2_no_anomaly['class'] == 'GALAXY']
qso = df2_no_anomaly[df2_no_anomaly['class'] == 'QSO']
star = df2_no_anomaly[df2_no_anomaly['class'] == 'STAR']

# Convert class labels to numerical values
le = LabelEncoder()
df2_no_anomaly["class"] = le.fit_transform(df2_no_anomaly["class"])
df2_no_anomaly["class"] = df2_no_anomaly["class"].astype(int)

# Prepare data for box plots
redshift = df2_no_anomaly[['redshift', 'class']]
data = [galaxy['redshift'], qso['redshift'], star['redshift']]
class_names = ['galaxy', 'qso', 'star']
colors = ['mediumslateblue', 'indianred', 'mediumaquamarine']

# Create the plot
fig, ax1 = plt.subplots(figsize=(9, 4))

bplot1 = ax1.boxplot(data,
                     vert=True,
                     patch_artist=True)

# Set x-tick labels using the class names
ax1.set_xticklabels(class_names)

# Fill boxes with colors
for patch, color in zip(bplot1['boxes'], colors):
    patch.set_facecolor(color)

# Add horizontal grid lines
ax1.yaxis.grid(True)
ax1.set_ylabel('Redshift')
plt.show()
```

```

for i in range(3):
    sns.kdeplot(data=np.log(df2_no_anomaly[
    df2_no_anomaly["class"] == i]['redshift']),
    label = le.inverse_transform([i]), color=
    colors[i])

Classes = ['GALAXY', 'QSO', 'STAR']
plt.legend(Classes)

from matplotlib.colors import to_rgba
data = [galaxy['u'], qso['u'], star['u']]
Classes = ['GALAXY', 'QSO', 'STAR']

colors = ['mediumslateblue', 'indianred', '
    mediumaquamarine']

my_colors = [to_rgba(c) for c in colors]

sns.set_palette(my_colors)

def kde(feature, loc):
    for i in range(3):
        sns.kdeplot(ax=loc, data=df2_no_anomaly[
    df2_no_anomaly["class"] == i][feature], label
    = le.inverse_transform([i]), color=colors[i])
        loc.grid()
        loc.set_xlabel(feature)
        loc.set_xlim([10, 30])
        loc.set_xticks(np.arange(10, 30, 2))
        loc.legend(Classes)


def box(feature, loc):
    data = [galaxy[feature], qso[feature], star[
    feature]]

    sns.boxplot(ax=loc, data=data)
    loc.set_xticklabels(Classes)
    loc.set_yticks(np.arange(10, 36, 2))
    loc.set_ylabel(feature)


fig, ax = plt.subplots(5, 2, figsize=(8, 16))#,
    sharex='col')

kde('u', ax[0, 1])
box('u', ax[0, 0])
kde('g', ax[1, 1])
box('g', ax[1, 0])
kde('r', ax[2, 1])
box('r', ax[2, 0])
kde('i', ax[3, 1])
box('i', ax[3, 0])
kde('z', ax[4, 1])
box('z', ax[4, 0])
plt.tight_layout()

from matplotlib.colors import to_rgba
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

data = [galaxy['u'], qso['u'], star['u']]
Classes = ['GALAXY', 'QSO', 'STAR']

colors = ['gray', 'mediumblue', 'green']  #
    Updated colors

my_colors = [to_rgba(c) for c in colors]

sns.set_palette(my_colors)

def kde(feature, loc):
    for i in range(3):
        sns.kdeplot(ax=loc, data=df2_no_anomaly[
    df2_no_anomaly["class"] == i][feature], label=
    le.inverse_transform([i]), color=colors[i],
    linewidth=2, linestyle='--')  # Changed line
    style
        loc.grid()
        loc.set_xlabel(feature)
        loc.set_xlim([10, 30])
        loc.set_xticks(np.arange(10, 30, 2))
        loc.legend(Classes)
        loc.spines['top'].set_visible(False)  #
    Hide top spine
        loc.spines['right'].set_visible(False)  #
    Hide right spine

def box(feature, loc):
    data = [galaxy[feature], qso[feature], star[
    feature]]

    sns.boxplot(ax=loc, data=data, palette=colors)
      # Use the same color palette
    loc.set_xticklabels(Classes)
    loc.set_yticks(np.arange(10, 36, 2))
    loc.set_ylabel(feature)
    loc.spines['top'].set_visible(False)
    loc.spines['right'].set_visible(False)
    loc.yaxis.grid(True)

fig, ax = plt.subplots(5, 2, figsize=(10, 18))

kde('u', ax[0, 1])
box('u', ax[0, 0])
kde('g', ax[1, 1])
box('g', ax[1, 0])
kde('r', ax[2, 1])
box('r', ax[2, 0])
kde('i', ax[3, 1])
box('i', ax[3, 0])
kde('z', ax[4, 1])
box('z', ax[4, 0])

plt.tight_layout()
plt.show()

df_no_anomaly.head()

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# Fit the scaler on the features and transform
    them
df_no_anomaly_scaled = pd.DataFrame(scaler.
    fit_transform(df_no_anomaly))

df_no_anomaly_scaled.head()

import pandas as pd
from sklearn.model_selection import
    train_test_split
# Separate features (X) and target (y)

X = df_no_anomaly_scaled.drop(columns=[7])
y = df_no_anomaly['class']
# Split the dataset into training and test sets,
    stratifying based on 'y'
X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.2, stratify
    =y, random_state=42)
X.head()

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

# Create a kNN model
```

```
k = 5  # Number of neighbors
knn_model = KNeighborsClassifier(n_neighbors=k)

# Fit the kNN model on the training data
knn_model.fit(X_train, y_train)

X_train.head()

# Make predictions on the test data
y_pred = knn_model.predict(X_test)

# Evaluate the model
accuracy = knn_model.score(X_test, y_test)
print("Accuracy:", accuracy)

# Generate a classification report
report = classification_report(y_test, y_pred)
print("Classification Report:\n", report)

'''''from sklearn.model_selection import
    cross_val_score
# Test different values of k
k_values = range(1, 15)
cv_scores = []

# Calculate cross-validation scores for each k
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train
    , cv=5, scoring='accuracy')
    cv_scores.append(scores.mean())

# Plot the model complexity curve
plt.figure(figsize=(8, 6))
plt.plot(k_values, cv_scores, marker='o')
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Cross-Validation Accuracy")
plt.title("accuracy vs number of neighbours")
plt.show()

X.head()

X_train.head()

# Custom weights for attributes
attribute_weights = [1,1,1,1,1,1,4]

# Instantiate KNeighborsClassifier with custom
    weights
k = 5  # Number of neighbors
knn_classifier_weighted = KNeighborsClassifier(
    n_neighbors=k, weights='distance', metric='
    minkowski', p=2, metric_params={'w':
    attribute_weights})

# Fit the classifier to your data
knn_classifier_weighted.fit(X_train, y_train)

# Evaluate the model
accuracy2 = knn_classifier_weighted.score(X_test,
    y_test)
print("Accuracy:", accuracy2)

# Make predictions on the test data
y_pred_2 = knn_classifier_weighted.predict(X_test)

# Generate a classification report
report2 = classification_report(y_test, y_pred_2)
print("Classification Report:\n", report2)
```