AI homework – Walid Alhashimi, Bruno Fares, Michael Rahme.

Exploratory data analysis:

Extracting the ARFF File Using Scipy and inspecting it:

Once the file is downloaded, we use the **arff** module from **Scipy** to load and parse the ARFF file. The ARFF file typically consists of metadata (attributes) followed by data (instances).

Code:

```
from scipy.io import arff
import pandas as pd

# Load the ARFF file
data, meta = arff.loadarff('filename.arff')

# Convert the data to a Pandas DataFrame
df = pd.DataFrame(data)
```

2- Checking for missing values:

In this step, we will check if there are any missing values in the dataset. Missing data can occur in various ways, such as empty cells, NaN values, or placeholders like None. It's crucial to identify and handle missing values appropriately before proceeding with any further analysis or modeling. We will use the **Pandas** library to check for missing values in the dataset.

Code:

```
# Check for missing values in the entire dataset

missing_values = df.isnull().sum()

# Display the number of missing values for each column

print("Missing values in each column:\n", missing_values)
```

Result: buying 0 / maint 0 / doors 0 / persons 0 / lug_boot 0 / safety 0/ class 0

There are 0 missing values in this data set.

2-Basic description:

Code:
```
#print the meta data extracted from the arff file that shows the categories and their values
print(meta)
```

Result: The car_df dataset contains several features related to car attributes and ratings. All features in this dataset are **nominal** (categorical), and they represent different aspects of the cars being analyzed. The features are as follows:

1. **buying**:
    - **Type**: Nominal
    - **Range**: ('vhigh', 'high', 'med', 'low')
    - This feature represents the buying price of the car, where vhigh stands for very high, high for high, med for medium, and low for low.
2. **maint**:
    - **Type**: Nominal
    - **Range**: ('vhigh', 'high', 'med', 'low')
    - This feature represents the maintenance cost of the car, with similar ranges as the buying feature. It categorizes the cost of maintaining the car as vhigh, high, med, or low.
3. **doors**:
    - **Type**: Nominal
    - **Range**: ('2', '3', '4', '5more')
    - This feature indicates the number of doors on the car. The possible values are 2, 3, 4, or 5more, which represent the number of doors, with 5more indicating a car with more than 5 doors.
4. **persons**:
    - **Type**: Nominal
    - **Range**: ('2', '4', 'more')
    - This feature describes the capacity of the car in terms of how many persons it can accommodate. The possible values are 2, 4, or more, with more indicating cars that can carry more than 4 people.
5. **lug_boot**:
    - **Type**: Nominal
    - **Range**: ('small', 'med', 'big')
    - This feature represents the size of the car's luggage boot (trunk). The possible values are small, med (medium), and big.
6. **safety**:
    - **Type**: Nominal
    - **Range**: ('low', 'med', 'high')
    - This feature indicates the safety level of the car, where low, med, and high refer to different safety ratings.
7. **class**:
    - **Type**: Nominal
    - **Range**: ('unacc', 'acc', 'good', 'vgood')
    - This is the target feature of the dataset and represents the classification of the car based on its attributes. The possible values are:
        - unacc: unacceptable
        - acc: acceptable
        - good: good
        - vgood: very good

Since the dataset contains categorical features (such as buying, maint, doors, etc.), machine learning models require numerical input. Therefore, we applied **encoding** techniques to transform these categorical variables into numerical form.

Code:

```
# Importing the LabelEncoder from sklearn.preprocessing
from sklearn.preprocessing import LabelEncoder

# Initialize the LabelEncoder object
label_encoder = LabelEncoder()

# Loop through each column in the dataset
for column in df.columns:
        # Apply Label Encoding to each column
         # This converts categorical values in each column to numeric values
        df[column] = label_encoder.fit_transform(df[column])
```

Class Distribution:

Code:

```
# Check the percentage distribution of each encoded class
class_distribution = df['class'].value_counts(normalize=True) * 100

 # Convert the encoded values back to the original class labels
class_distribution =
class_distribution.rename(index=dict(enumerate(label_encoder.classes_)))

# Print the distribution in percentages
 print("Class distribution (percentage):\n", class_distribution)
```

Result: After analyzing the dataset, we observed the following distribution of the target variable class:

- **Unacceptable (unacc)**: 70.02%
- **Acceptable (acc)**: 22.22%
- **Good (good)**: 3.99%
- **Very Good (vgood)**: 3.76%

This indicates a **high imbalance** in the dataset, with the majority of instances classified as **unacceptable** (unacc), and only a small portion of the data representing **good** or **very good** classifications. This imbalance is important to consider, as it could affect the performance of machine learning models, especially when evaluating classification accuracy.

3-Identifying the shape of the data:

Code:

```
# Get the shape of the DataFrame
shape = df.shape

# Print the shape
print(f"The dataset contains {shape[0]} rows and {shape[1]} columns.")
```

Result:

The dataset contains 1728 rows and 7 columns. It's a relatively small dataset.

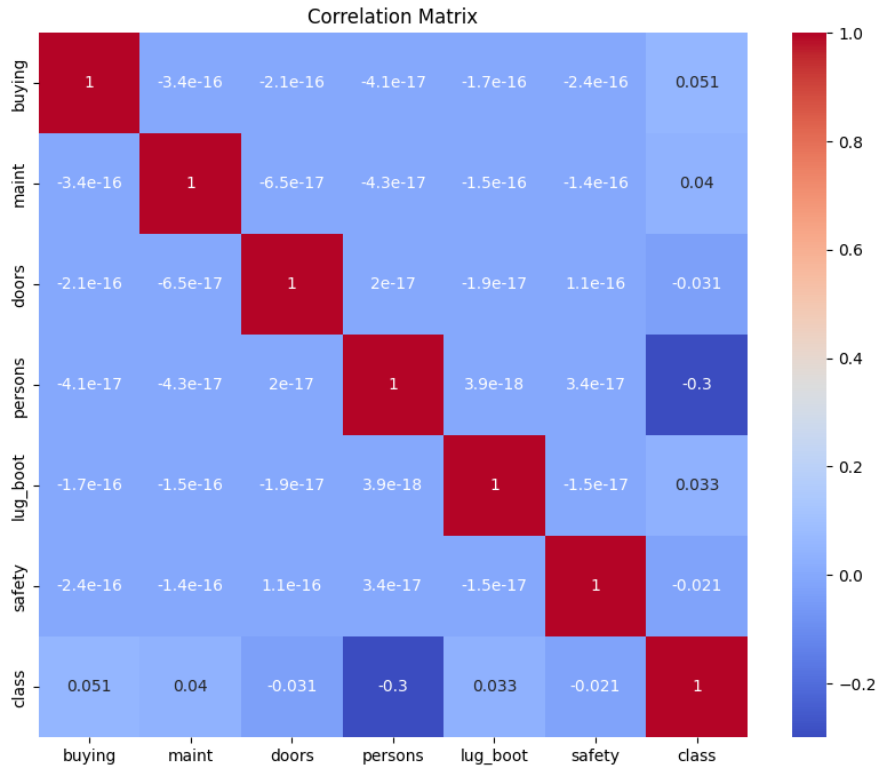
4-Indetifying significant correlations:

Correlation Analysis: In this section, we analyze the relationships between the features using a **correlation matrix**. As the dataset contains categorical variables. After encoding, we used **Seaborn** to visualize the correlation matrix with a heatmap:

Code:

```
# Generate the correlation matrix
plt.figure(figsize=(10,8))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

Result:



The correlation values were found to be **close to 0** across the features, suggesting there is no strong linear relationship between the variables. This result is likely due to the nature of the encoded categorical data, as the encoding assumes an ordinal relationship that may not exist. Therefore, the Pearson correlation might not be the best measure for categorical features.

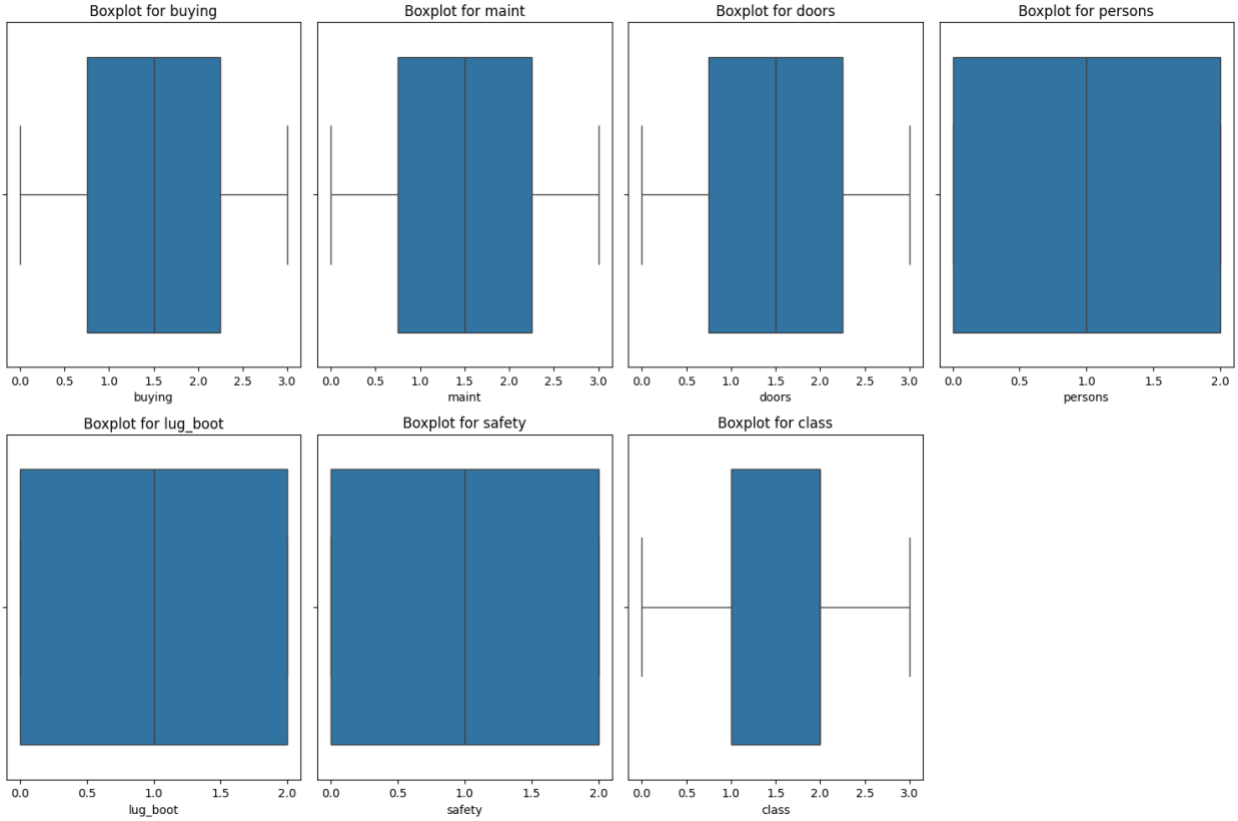
5-Outliers:

Outliers are typically associated with numerical data, but in this case, we applied a **boxplot** to check for outliers, even though it is primarily designed for continuous variables.

Code:

```
#generate a box plot for every column
for col in df.columns:
        plt.figure(figsize=(6,4))
        sns.boxplot(x=df[col])
        plt.title(f'Boxplot for {col}')
        plt.show()
```

Result: No outliers, with a box plot generated for each category:



Splitting the dataset:

The dataset is split 80% for training and 20% for testing, we chose a random seed (42), however we kept using this random number so that the results can be reproduced for different models:

Code:

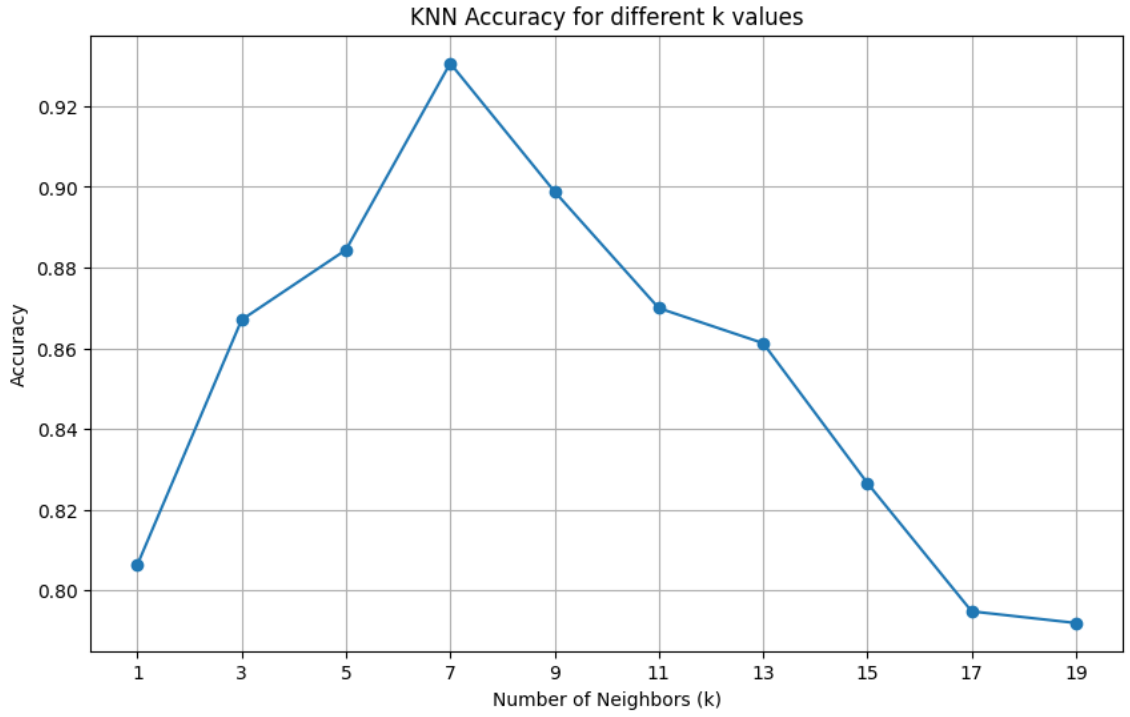
```
X = df.drop('class',axis=1)
y = df['class']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Testing Models:

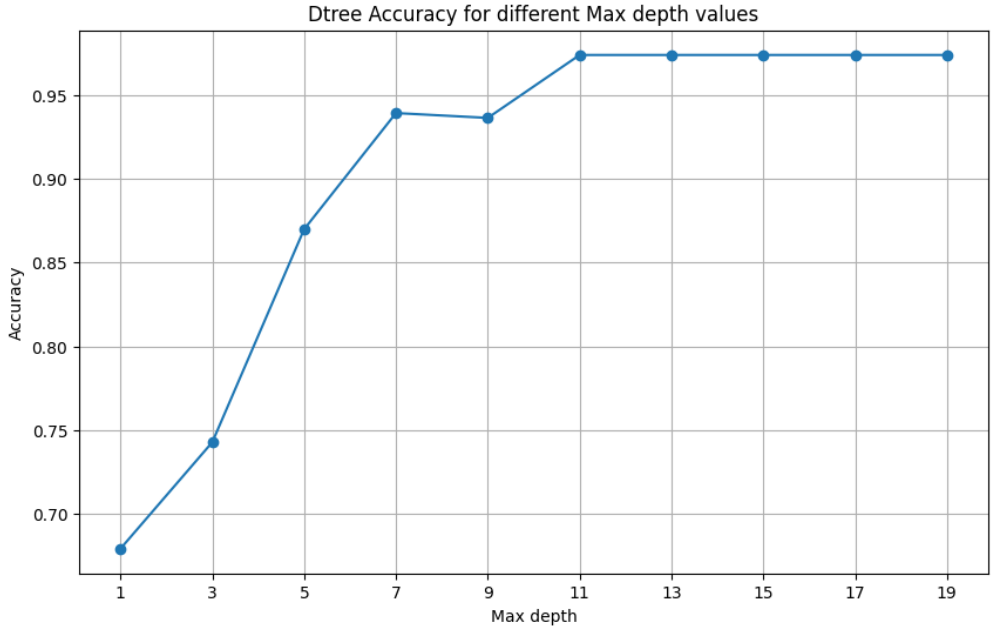
K-nearest neighbor:

We evaluated KNN using odd values of *k* from 1 to 19. The model achieved the highest accuracy of **93.06%** at **k=7**, indicating that moderate neighborhood sizes generalize best for this dataset. Accuracy decreased at higher *k* values, likely due to oversmoothing and including more distant, less relevant neighbors.



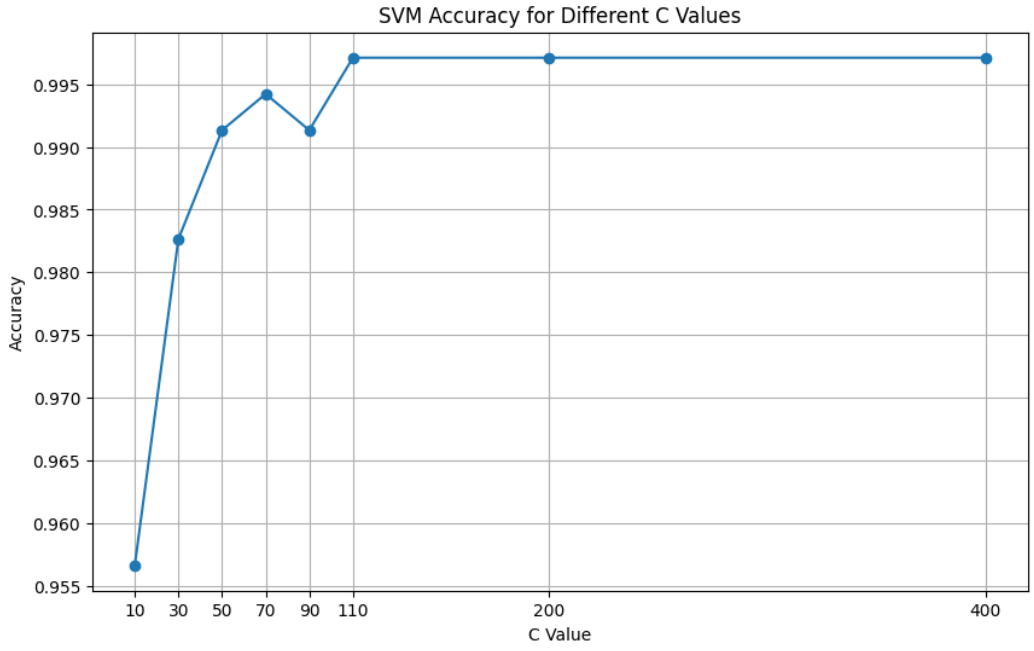
Decision Tree:

The Decision Tree model was evaluated using varying max_depth values from 1 to 19. Accuracy increased steadily with depth, peaking at **97.40%** for depths **11 and above**, suggesting the tree captures sufficient complexity by that point. Lower depths underfit the data, while deeper trees did not lead to overfitting in this case.



SVM:

After fine-tuning `C` across a narrower range, we observed steady accuracy improvements and decreasing hinge loss. Accuracy rose from **95.66% at C=10** to a peak of **99.71% at C=110 and beyond**, while hinge loss dropped below **0.005**, indicating highly confident and correct predictions. Performance plateaued beyond `C=110`, suggesting diminishing returns for higher values.



Conclusion:

After evaluating all three models on the same dataset:

- **K-Nearest Neighbors (KNN)** achieved its highest accuracy of **93.06%** at **k=7**.
- **Decision Tree** reached a peak accuracy of **97.40%** at **max_depth ≥ 11**.
- **Support Vector Machine (SVM)** achieved the highest performance overall, with an accuracy of **99.71%** at **C = 110 and above**, along with minimal hinge loss.

Among the three models, **SVM outperformed both KNN and Decision Tree**, making it the most effective classifier for this dataset in terms of accuracy and confidence.