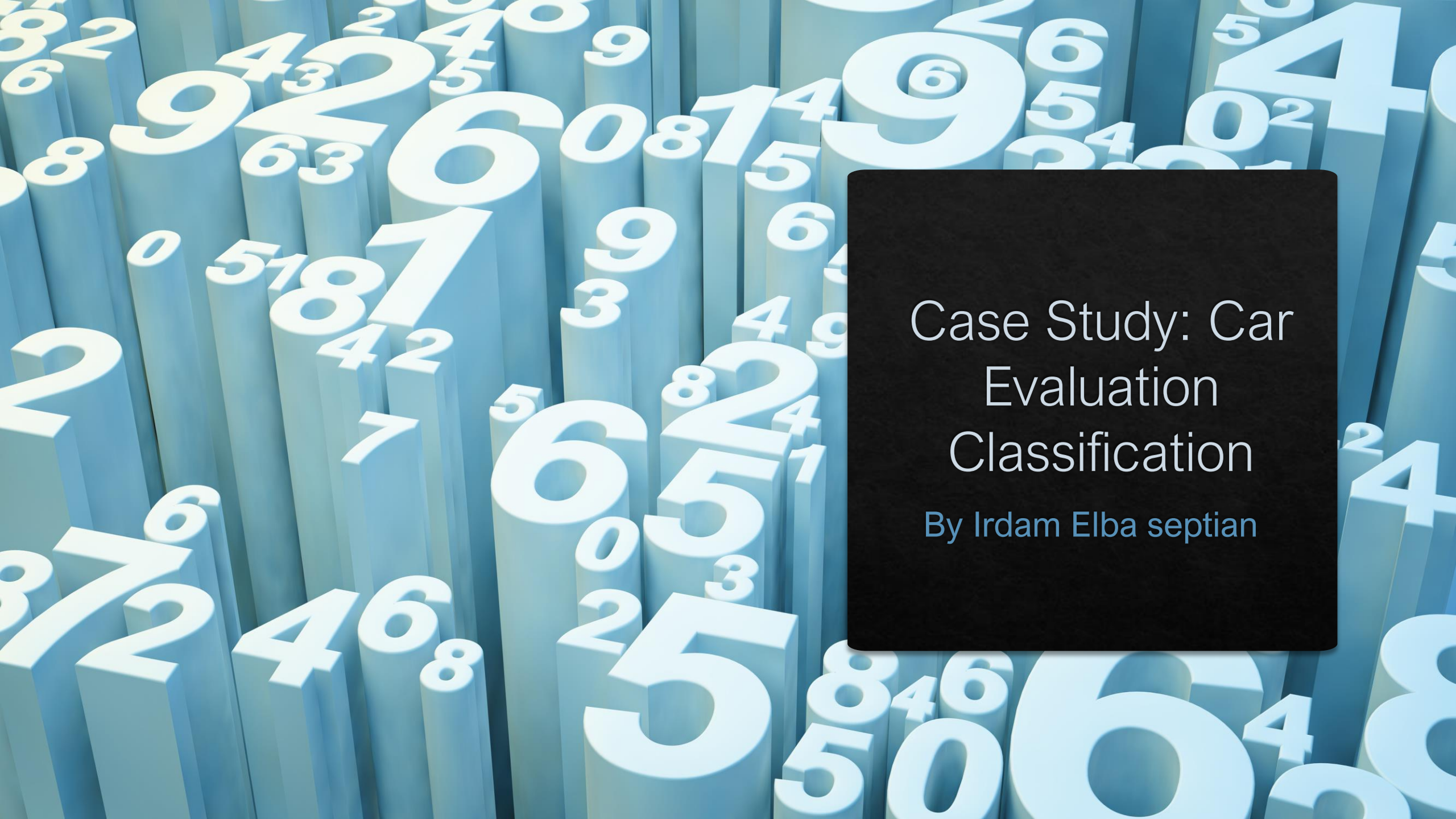


Case Study: Car Evaluation Classification

By Irdam Elba septian

TABLE OF CONTENTS

INTRODUCTION

Hello! My name is Irdam Elba Septian and I am a student in Business Intelligence batch 12 at dibimbing.id. In this slides I'm about to share my mini project with you. I will be using tools such as Google Colab to code in Python in order to make machine learning model, in this case is classification.

INTRODUCTION

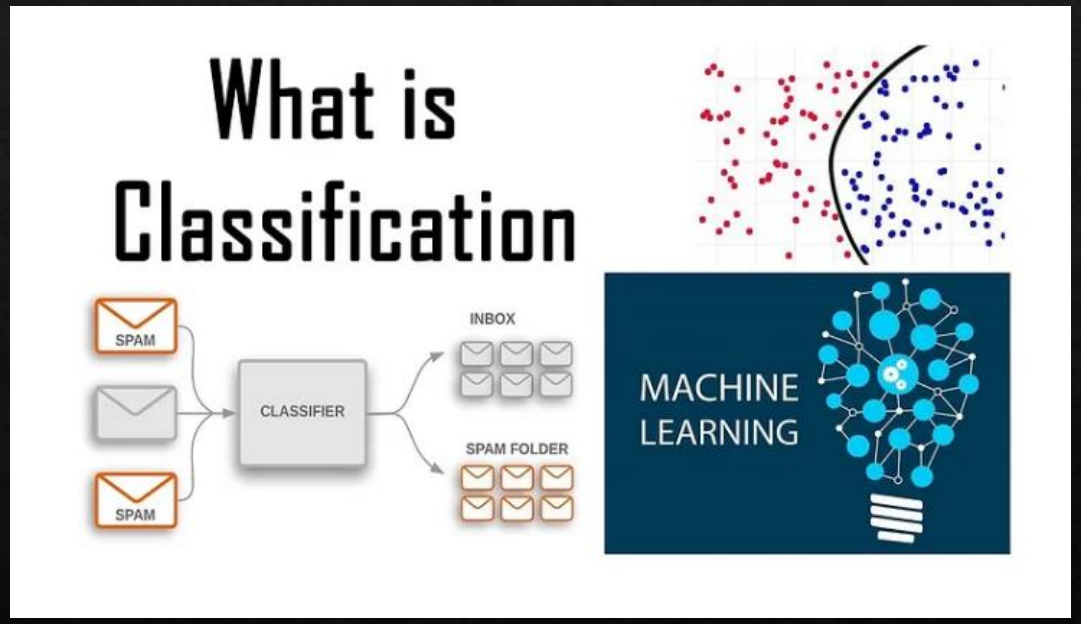
- In this example, the dataset is focused on car evaluations. It is designed to determine car quality based on several key factors.
- I will apply any necessary data transformations to the retrieved data.
- Finally, I will perform classification using machine learning in this example.
- Classification is a type of machine learning task where the goal is to categorize data into predefined classes or labels.
- In a car evaluation dataset, classification involve predicting a car's quality as "unacceptable," "acceptable," "good," or "very good" based on various features such as price, maintenance cost, and safety.
- If you want to see the full coding, you can click here

What is a Google Colaboratory?

- Google Colaboratory is a cloud-based platform that allows users to write and execute Python code directly in their web browser. It provides an environment for machine learning, data analysis, and other computational tasks, offering free access to GPUs and TPUs.
- Google Colab is widely used for collaborative coding, especially for data science and artificial intelligence projects, as it enables users to share and execute notebooks seamlessly.

What is a Classification?

- Classification is a supervised machine learning technique that involves training a model to predict a categorical label (class) based on input data.
- In classification, each data point is associated with a specific label or category, and the goal is for the model to learn the patterns and relationships between the input features and these labels.
- Once trained, the model can then classify new, unseen data into one of the predefined categories.



Data Preprocessing

- Data preprocessing is a crucial step in data analysis and machine learning that involves preparing raw data to make it suitable for analysis.
- In this dataset, minimal data preprocessing has been performed, as the dataset is already well-structured and does not contain significant flaws.
- The majority of data preprocessing in this dataset involves replacing ambiguous values, checking for null values, and identifying duplicate entries.

Data Preprocessing

replacing ambiguous values

```
# checking unique value in every feature column
for col in x.columns:
    print(col, x[col].unique())
```

```
buying ['vhigh' 'high' 'med' 'low']
maint ['vhigh' 'high' 'med' 'low']
doors ['2' '3' '4' '5more']
persons ['2' '4' 'more']
lug_boot ['small' 'med' 'big']
safety ['low' 'med' 'high']
```

```
# checking unique values in target column
for col in y.columns:
    print(col, y[col].unique())
```

```
class ['unacc' 'acc' 'vgood' 'good']
```

```
# replace ambiguous value to be clearer
x['doors'] = x['doors'].replace('5more', 'more')
```

```
# checking again to see if its clear
for col in x.columns:
    print(col, x[col].unique())
```

```
buying ['vhigh' 'high' 'med' 'low']
maint ['vhigh' 'high' 'med' 'low']
doors ['2' '3' '4' 'more']
persons ['2' '4' 'more']
lug_boot ['small' 'med' 'big']
safety ['low' 'med' 'high']
```

Data Preprocessing

Checking null & duplicate

```
x.isnull().sum()
```

	0
buying	0
maint	0
doors	0
persons	0
lug_boot	0
safety	0

dtype: int64

```
y.isnull().sum()
```

	0
class	0

```
x.duplicated().sum()
```

```
0
```

Feature Engineering

- Feature engineering is the process of transforming raw data into meaningful features that can be used to improve the performance of a machine learning model.
- In this dataset, all columns contain categorical data, and, upon closer inspection, each column has an ordinal categorical data type
- The ordinal data type refers to categorical data where the categories have a specific order or ranking.
- As shown previously, the column values consist of categories like 'vhigh', 'high', 'med', 'low' or 'small', 'med', 'big'. This type of data has an inherent ranking, making it ordinal data.

Feature Engineering

- One of the most important steps in the machine learning process is encoding categorical data. Encoding involves converting categorical values into numerical format so that machine learning models can interpret and learn from the data, as they can only process numerical inputs.
- Since we are working with ordinal data, it is preferable to use label encoding.
- Label encoding converts categorical data into numerical values by assigning a unique integer to each category. It is particularly useful for ordinal data, as it preserves the natural order of categories.
- In this case, I manually labeled the data because each column has a small number of unique values, and automatic labeling sometimes randomly assigns numbers.

Feature Engineering

Label Encoding Process

Before

	buying	maint	doors	persons	lug_boot	safety
0	vhigh	vhigh	2	2	small	low
1	vhigh	vhigh	2	2	small	med
2	vhigh	vhigh	2	2	small	high
3	vhigh	vhigh	2	2	med	low
4	vhigh	vhigh	2	2	med	med

```python
# encoding categorical column that have ordinal value
lug_boot_mapping = {'big': 3, 'med': 2, 'small': 1}
safety_mapping = {'high': 3, 'med': 2, 'low': 1}
buying_maint_mapping = {'vhigh': 4, 'high': 3, 'med': 2, 'low': 1}
doors_mapping = {'2': 1, '3': 2, '4': 3, 'more': 4}
persons_mapping = {'2': 1, '4': 2, 'more': 3}

x['lug_boot'] = x['lug_boot'].map(lug_boot_mapping)
x['safety'] = x['safety'].map(safety_mapping)
x['buying'] = x['buying'].map(buying_maint_mapping)
x['maint'] = x['maint'].map(buying_maint_mapping)
x['doors'] = x['doors'].map(doors_mapping)
x['persons'] = x['persons'].map(persons_mapping)
```

After

```python
x.head()
```

	buying	maint	doors	persons	lug_boot	safety
0	4	4	1	1	1	1
1	4	4	1	1	1	2
2	4	4	1	1	1	3
3	4	4	1	1	2	1
4	4	4	1	1	2	2

TRAINING & TESTING

- Training in machine learning is the process where the model learns patterns from labeled data and optimizes its parameters.
- After training, testing is performed using unseen data to evaluate how well the model generalizes and makes accurate predictions on new data.
- In this dataset, we split the data into 70% for training and 30% for testing.
- I separate the data into 70% for training and 30% for testing to ensure that the model has enough data to learn from while also being evaluated on a separate, unseen dataset. The 70% training data allows the model to learn patterns and adjust its parameters, while the 30% testing data is used to assess how well the model generalizes to new data.

TRAINING & TESTING

The process splitting the data

```
from sklearn.model_selection import train_test_split

# Splitting the data into training and testing sets (for example, 70% for training and 30% for testing)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42)

print("Train Data:")
print(x_train.shape, y_train.shape)

print("Test Data:")
print(x_test.shape, y_test.shape)
```

```
Train Data:
(1209, 6) (1209, 1)
Test Data:
(519, 6) (519, 1)
```

TRAINING & TESTING

- A classification model is a type of machine learning model used to predict categorical labels or classes for given input data.
- In classification tasks, the goal is to assign a new observation to one of several predefined classes based on patterns learned from a labeled dataset.
- In Python, there are libraries that provide pre-built classification models, enabling you to apply them directly and let the model perform the task of classifying data
- In this case, the model I use is *Random Forest* and *XGBoost*

TRAINING & TESTING

- Before we look at the results, let's review the metrics commonly used in classification.
  - Accuracy: The percentage of correct predictions out of all predictions, showing overall model effectiveness. However, it may be misleading on imbalanced datasets.
  - Precision: The proportion of true positive predictions out of all positive predictions, important when false positives need minimization.
  - Recall (Sensitivity): The proportion of true positives correctly identified, crucial in scenarios where missing a positive case is costly.
  - F1-Score: The harmonic mean of precision and recall, balancing both metrics for datasets with class imbalance.

TRAINING & TESTING

Random Forest Classification process

```
# using random forest as the model
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

model_rf = RandomForestClassifier(random_state=42)
model_rf.fit(x_train, y_train)
y_pred = model_rf.predict(x_test)
```

As shown in the image, we import the Random Forest classifier from a Python library to easily train and apply the model.

TRAINING & TESTING

Random Forest Classification process

```
# show classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))

# show confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

         acc       0.95      0.92      0.94       118
        good       0.74      0.89      0.81        19
       unacc       1.00      0.99      1.00       358
       vgood       0.83      0.83      0.83        24

    accuracy                           0.97       519
   macro avg       0.88      0.91      0.89       519
weighted avg       0.97      0.97      0.97       519

Confusion Matrix:
[[109   5   1   3]
 [  1  17   0   1]
 [  2   0 356   0]
 [  3   1   0  20]]
```

The classification report shows that the model performs well overall, with high accuracy (97%) and excellent precision, recall, and F1-scores for the 'unacc' class, which is the most frequent. However, performance on the minority classes like 'good' and 'vgood' is lower, with F1-scores of 0.81 and 0.83, respectively. The confusion matrix reveals that the model correctly predicts the majority class ('unacc') most of the time but misclassifies a few instances of 'good' and 'vgood'. Despite these misclassifications, the model is still effective, particularly for the dominant class.

TRAINING & TESTING

XGBoost Classification process

```
# using xgboost model
import xgboost as xgb
from xgboost import XGBClassifier

# mapping target column because xgboost only works on numeric
y_mapping = {'unacc': 0, 'acc': 1, 'good': 2, 'vgood': 3}
y_encoded = y['class'].map(y_mapping)
```

```
# Splitting the data into training and testing sets (for example, 70% for training and 30% for testing)
x_train, x_test, y_train, y_test = train_test_split(x, y_encoded, test_size=0.3, random_state=42)

# initialize the xgboost model
model = xgb.XGBClassifier()

# train model
model.fit(x_train, y_train)

# Predict
y_pred = model.predict(x_test)
```

As shown in the image, I used the same method as in Random Forest, but I made a few adjustments in XGBoost.

TRAINING & TESTING

XGBoost Classification process

```
# Evaluating model
print("Accuracy on Test Data:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))

# showing confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

```
Accuracy on Test Data: 0.9788053949903661
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       358
           1       0.97      0.96      0.97       118
           2       0.74      0.89      0.81        19
           3       0.91      0.88      0.89        24

    accuracy                           0.98       519
   macro avg       0.91      0.93      0.92       519
weighted avg       0.98      0.98      0.98       519

Confusion Matrix:
[[357   1   0   0]
 [  0 113   4   1]
 [  0   1  17   1]
 [  0   1   2  21]]
```

The model demonstrates excellent performance with an overall accuracy of 97.88%. In the classification report, precision, recall, and F1-score for the majority class (Class 0) reach 1.00, indicating nearly perfect predictions. Other classes, such as Class 1, also have high scores (F1-score of 0.97), while smaller classes like Class 2 have slightly lower scores (F1-score of 0.81). The confusion matrix confirms these results, with most predictions being accurate across all classes. Overall, the model is highly accurate, especially on the majority class, with minor misclassifications in the minority classes.

CONCLUSSION

- Using Random Forest, the model achieves an overall accuracy of 97%, with high scores across all metrics. Looking at precision and recall, the model performs exceptionally well on the "unacc" class, where it achieves a true positive rate close to 100% (precision of 1.00 and recall of 0.99), indicating minimal false positives and false negatives. For the "acc" class, the recall is 0.92, meaning most true positives are identified, though some are missed (false negatives). However, for the "good" class, the model has lower precision (0.74), indicating a higher number of false positives, but good recall (0.89), showing most actual positives are correctly identified. F1-scores for each class show balanced performance, especially for the "unacc" (1.00) and "acc" (0.94) classes, but are slightly lower for the "good" (0.81) and "vgood" (0.83) classes.

CONCLUSSION

- Using XGBoost, the model achieves a slightly higher accuracy of 98%. Precision and recall remain perfect for the "unacc" class (both 1.00), meaning there are zero false positives and false negatives for this major class. The "acc" class also shows high precision (0.97) and recall (0.96), indicating a low rate of false positives and negatives. For the minority "good" and "vgood" classes, precision and recall are consistent with the first model (precision of 0.74 and recall of 0.89 for "good", and recall of 0.88 for "vgood"), demonstrating that this model captures most true positives in these classes with minimal false positives. F1-scores are high across classes, with improvements seen in "vgood" (0.89). The macro average scores are slightly higher (0.91 precision, 0.93 recall, 0.92 F1), showing better handling of class balance.

***notes: if you want to know more about the metrics, you can go here*

CONCLUSSION

- In comparison, both models perform well, but the *XGBoost* has marginally higher macro and weighted averages, indicating better balance across all classes. The lower rate of false positives and false negatives, especially in the "acc" and "unacc" classes, contributes to the slight improvement in overall performance in the model.
- In this dataset, *XGBoost* could work better due to its boosting technique, which helps improve accuracy and handle class imbalance more effectively. *XGBoost* achieves higher overall accuracy (98% vs. 97%) and better F1-scores, particularly for minority classes like "vgood," where it balances precision and recall more effectively. The sequential correction of errors in *XGBoost* allows it to focus on harder-to-classify instances, leading to improved performance compared to Random Forest.

THANK YOU