FACULTY OF ENGINEERING TECHNOLOGY

COMPUTER SCIENCE DEPARTMENT

COMP4388, MACHINE LEARNING
ASSIGNMENT 1

Prepared by:

Zaid Nedal Mousa 1221833

Instructor: Dr. Radi Jarrar

Contents

Table of Figures

Task 1

1. EDA

```python
# Set pandas options to display all columns and rows without truncation
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
pd.set_option('display.float_format', '{:.2f}'.format)

# Exploratory Data Analysis (EDA)
print("Summary Statistics:")
print(data.describe())
```

Figure 1: EDA code

```
Summary Statistics:
          ID  Call Failure  Charge Amount  Freq. of use  Freq. of SMS  Distinct Called Numbers  Age Group     Age  Customer Value
count 3150.00       3150.00        3150.00       3150.00       3150.00                  3150.00    3150.00 3150.00         3150.00
mean  1575.50          7.63         129.88         69.46         73.17                    23.51       2.83   31.00          470.97
std    909.47          7.26         102.79         57.41        112.24                    17.22       0.89    8.83          517.02
min      1.00          0.00          20.00          0.00          0.00                     0.00       1.00   15.00            0.00
25%    788.25          1.00          50.00         27.00          6.00                    10.00       2.00   25.00          113.80
50%   1575.50          6.00         100.00         54.00         21.00                    21.00       3.00   30.00          228.48
75%   2362.75         12.00         200.00         95.00         87.00                    34.00       3.00   30.00          788.39
max   3150.00         36.00         400.00        255.00        522.00                    97.00       5.00   55.00         2165.28
```

Figure2:EDA statistics

This summarize the data and describe it and hive some information about the(count of them, mean, Standard Deviation, min and max, and the quartiles)

2. Distribution for class label (churn)

```
print("\nClass Label Distribution (Churn):")
print(data['Churn'].value_counts())

data['Churn'].value_counts().plot(kind='bar', title='Distribution of Churn')
plt.xlabel('Churn')
plt.ylabel('Frequency')
plt.show()
```

Figure 3: churn distribution code

```
Class Label Distribution (Churn):
Churn
yes    2655
no      495
```

Figure 4: Distribution on Terminal

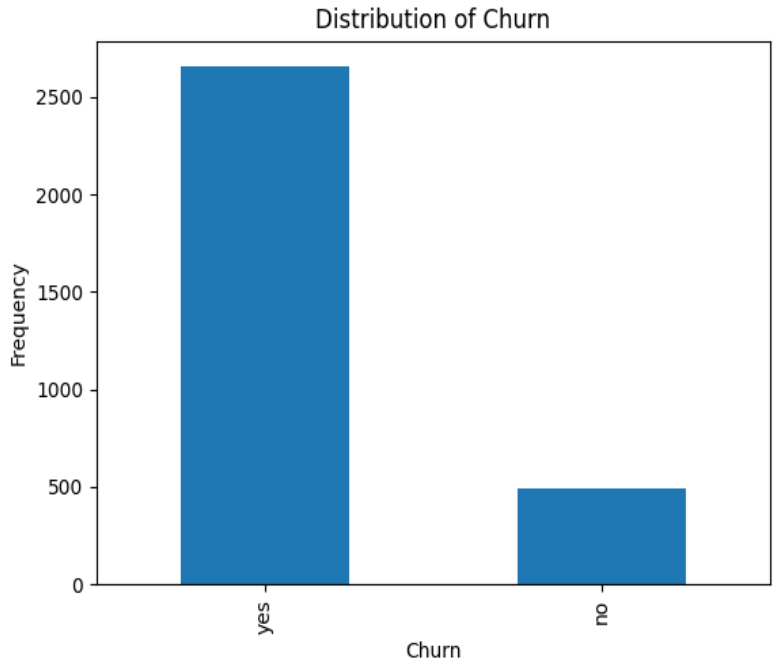

Figure 5: Distribution plot

These shows the Distribution for the class label **Churn**

3. Amount of churn in each sub-group in age group

```
# Histograms for Age Group and Charge Amount
sns.histplot(data=data, x='Age Group', hue='Churn', multiple='dodge', shrink=0.8)
plt.title('Churn by Age Group')
plt.show()
```

Figure 6:Histogram details change code

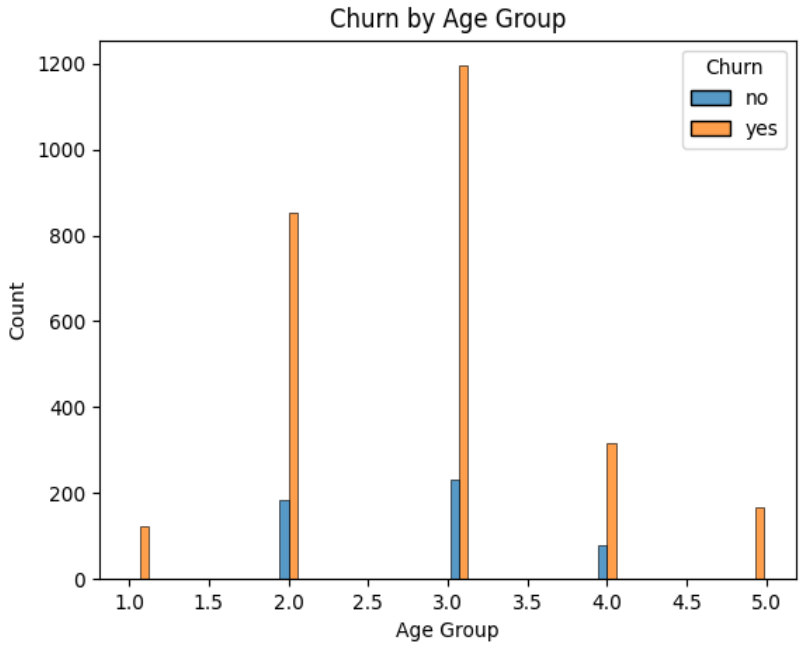

Figure 7:Histogram for details churn with age group

4. Amount of Churn in each sub-group of Charge Amount

```
sns.histplot(data=data, x='Charge Amount', hue='Churn', multiple='dodge', shrink=0.8)
plt.title('Churn by Charge Amount')
plt.show()
```

Figure 8:Histogram details charge amount and churn code

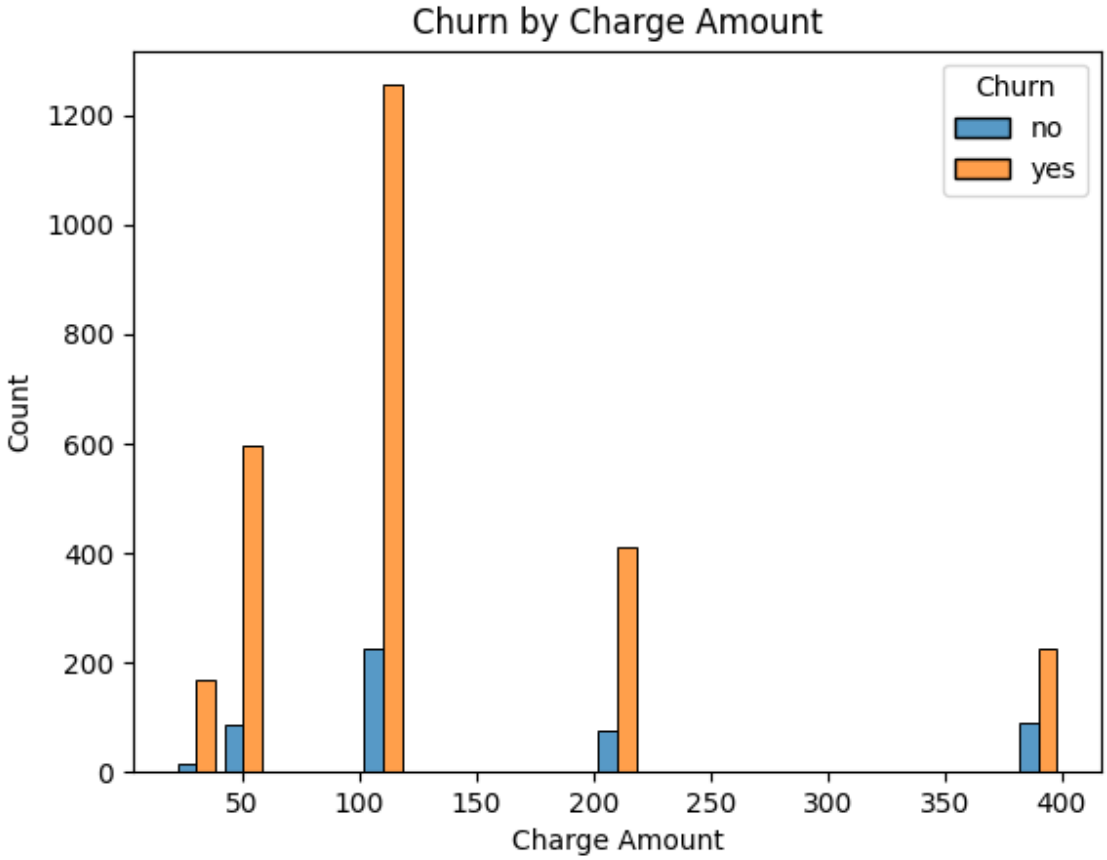

Figure 9:Histogram for details churn with charge amount

5. Details of charge amount

```
print("Charge Amount Statistics:")
print(data['Charge Amount'].describe())
```

Figure 10: charge amount details code

```
Charge Amount Statistics:
count   3150.00
mean     129.88
std      102.79
min       20.00
25%       50.00
50%      100.00
75%      200.00
max      400.00
```

Figure 11: charge amount details

6. Correlation between features

```
# Handle missing values explicitly for numeric columns
numeric_cols = data.select_dtypes(include=[np.number]).columns
imputer = SimpleImputer(strategy='mean')  # Replace missing values with the mean
data[numeric_cols] = imputer.fit_transform(data[numeric_cols])

# Handle missing values for non-numeric columns explicitly
data['Plan'] = data['Plan'].fillna('unknown')  # Replace missing values in 'Plan' with 'unknown'

# Check for missing values again
print("Number of missing values after handling:", data.isna().sum().sum())

# Convert binary attributes to numeric
data['Churn'] = data['Churn'].map({'yes': 1, 'no': 0})  # Convert Churn to binary
data['Plan'] = data['Plan'].map({'pre-paid': 0, 'post-paid': 1, 'unknown': -1})  # Convert Plan to binary or handle 'unknown'
data['Complains'] = data['Complains'].map({'yes': 1, 'no': 0})  # Example for binary encoding
data['Status'] = data['Status'].map({'active': 1, 'not-active': 0})  # Example for binary encoding

# Correlation Analysis
correlation_matrix = data.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

Figure 12: correlation features code

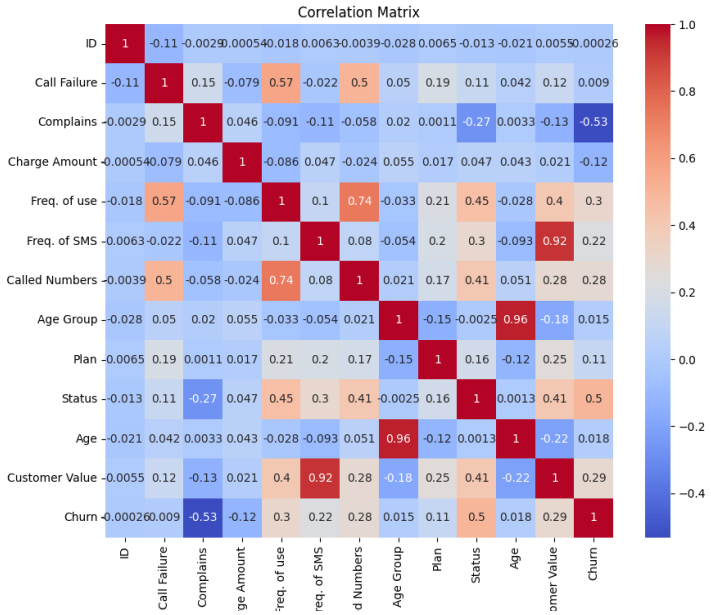

Figure 13: correlation matrix

From the matrix, it is clear that Customer Value, Freq. of use, and Complains have significant relationships with other features and are particularly relevant for predicting churn. These features should be prioritized when developing a churn prediction model.

On the other hand, features with weak correlations, such as Plan, Status, and Call Failure, may not contribute much to the prediction and might either require transformation to improve their relevance or could be excluded altogether to simplify the model and enhance its performance. This approach ensures that the focus remains on the most impactful variables.

7. Split Data (Training and Test)

```
# Selecting Features based on Correlation
selected_features_all = ['ID', 'Call Failure', 'Complains', 'Charge Amount', 'Freq. of use', 'Freq. of SMS',
                         'Distinct Called Numbers', 'Age Group', 'Plan', 'Status', 'Age']
X_all = data[selected_features_all]
y = data['Customer Value']

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split( *arrays: X_all, y, test_size=0.3, random_state=42)
```

Figure 14:split data code

Here we just split the data to: Training data 70% and Test Data 30%

Regression Task

1. Linear regression using all independent attributes

```
# LRM1: Linear Regression using all features
lr = LinearRegression()
lr.fit(X_train, y_train)
y_pred_lrm1 = lr.predict(X_test)

print("\nPerformance of LRM1 (All Features):")
print("MSE:", mean_squared_error(y_test, y_pred_lrm1))
print("R2 Score:", r2_score(y_test, y_pred_lrm1))
```

Figure 15:LRM1 Regression code

```
Performance of LRM1 (All Features):
MSE: 6875.757854367428
R2 Score: 0.9737774486449239
```

Figure 16: LRM1 performance

2. Linear regression using 2 most important features

```
# LRM2: Linear Regression using two selected features (Freq. of SMS and Freq. of use)
lr2 = LinearRegression()
lr2.fit(X_train[['Freq. of use', 'Freq. of SMS']], data['Customer Value'][X_train.index])
y_pred_lrm2 = lr2.predict(X_test[['Freq. of use', 'Freq. of SMS']])

print("\nPerformance of LRM2 (Two Features):")
print("MSE:", mean_squared_error(data['Customer Value'][X_test.index], y_pred_lrm2))
print("R2 Score:", r2_score(data['Customer Value'][X_test.index], y_pred_lrm2))
```

Figure 17: LRM2 Code

```
Performance of LRM2 (Two Features):
MSE: 11047.661633484944
R2 Score: 0.9578667718274066
```

Figure 18: LRM2 performance

We selected **Freq. of Use** and **Freq. of SMS** for LRM2 because they reflect customer engagement with the service, which directly influences customer value. High usage typically indicates a more valuable customer.

Freq. of Use reflects customer engagement and activity levels. Customers who frequently use the service are more likely to generate higher value. This engagement often translates to higher spending or longer retention.

Freq. of SMS: this feature complements it by capturing another dimension of service usage. A high number of SMS messages indicates frequent interaction with the telecommunication service, which may correlate with higher customer value.

3. Linear regression using set of important features

```
# LRM3: Linear Regression using top correlated features
top_features = ['Charge Amount', 'Freq. of use', 'Status', 'Age Group' , 'Freq. of SMS']  # Example based on correlation
lr3 = LinearRegression()
lr3.fit(X_train[top_features], data['Customer Value'][X_train.index])
y_pred_lrm3 = lr3.predict(X_test[top_features])

print("\nPerformance of LRM3 (Top Correlated Features):")
print("MSE:", mean_squared_error(data['Customer Value'][X_test.index], y_pred_lrm3))
print("R2 Score:", r2_score(data['Customer Value'][X_test.index], y_pred_lrm3))
```

Figure 19: LRM3 regression code

```
Performance of LRM3 (Top Correlated Features):
MSE: 7335.9803964012945
R2 Score: 0.972022266234656
```

Figure 20: LRM3 performance

We chose features based on their strong correlations with customer value, including **Charge Amount** (financial contribution), **Freq. of Use** and **Freq. of SMS** (engagement), **Status** (active or inactive), and **Age Group** (demographic trends). These features were chosen as key drivers of customer behavior and value, combining both domain insights and statistical significance.

Charge Amount reflects the financial contribution of the customer. Customers with higher charges are generally more valuable and engaged.

Freq. of Use: The correlation matrix may confirm that frequent usage of calls correlates strongly with customer value.

Status: Active customers are more likely to generate value for the company, while inactive ones contribute little to no revenue. This is an important predictor of churn and overall customer value.

Age Group: Customer value often varies by age. Younger customers may use the service differently than older ones, impacting their overall contribution to the business.

Freq. of SMS: The correlation matrix likely confirms that frequent SMS usage correlates strongly with customer value.

4. Compare between the performance for the

```
print("\nPerformance of LRM1 (All Features):")
print("MSE:", mean_squared_error(y_test, y_pred_lrm1))
print("R2 Score:", r2_score(y_test, y_pred_lrm1))
```

```
print("\nPerformance of LRM2 (Two Features):")
print("MSE:", mean_squared_error(data['Customer Value'][X_test.index], y_pred_lrm2))
print("R2 Score:", r2_score(data['Customer Value'][X_test.index], y_pred_lrm2))
```

```
print("\nPerformance of LRM3 (Top Correlated Features):")
print("MSE:", mean_squared_error(data['Customer Value'][X_test.index], y_pred_lrm3))
print("R2 Score:", r2_score(data['Customer Value'][X_test.index], y_pred_lrm3))
```

```
Performance of LRM1 (All Features):
MSE: 6875.757854367428
R2 Score: 0.9737774486449239

Performance of LRM2 (Two Features):
MSE: 11047.661633484944
R2 Score: 0.9578667718274066

Performance of LRM3 (Top Correlated Features):
MSE: 7335.9803964012945
R2 Score: 0.972022266234656
```

Figure 21: Comparing

Model	Features Used	MSE	R^2 Score
LRM1	All Independent Attributes	6875.75	0.973777
LRM2	Freq. of use, Freq. of SMS	11047.66	0.957866
LRM3	Charge Amount, Freq. of use, Status, Age Group, Freq. of SMS	7335.98	0.972022

Classification Task

1. KNN

```
# k-Nearest Neighbors
knn = KNeighborsClassifier()
knn.fit(X_train, data['Churn'][X_train.index])
y_prob_knn = knn.predict_proba(X_test)[:, 1]
```

Figure 22: KNN classification

2. Naive Bayes

```
# Naive Bayes
nb = GaussianNB()
nb.fit(X_train, data['Churn'][X_train.index])
y_prob_nb = nb.predict_proba(X_test)[:, 1]
```

Figure 23: Naive Bayes classification

3. Decision Tree

```
# Decision Tree
dt = DecisionTreeClassifier(random_state=42)
dt.fit(X_train, data['Churn'][X_train.index])
y_prob_dt = dt.predict_proba(X_test)[:, 1]
```

Figure 24: Decision Tree classification

4. ROC/AUC and Compare the performance between classification

```python
# Logistic Regression
log_reg = make_pipeline(
    *steps: StandardScaler(),
    LogisticRegression(max_iter=2000, random_state=42)
)
log_reg.fit(X_train, data['Churn'][X_train.index])
y_prob_lr = log_reg.predict_proba(X_test)[:, 1]
```

Figure 25: Logistic code

```python
# Accuracy for Logistic Regression
y_pred_lr = log_reg.predict(X_test)
accuracy_lr = accuracy_score(data['Churn'][X_test.index], y_pred_lr)
r2_lr = r2_score(data['Churn'][X_test.index], y_pred_lr)

# Accuracy for Decision Tree
y_pred_dt = dt.predict(X_test)
accuracy_dt = accuracy_score(data['Churn'][X_test.index], y_pred_dt)
r2_dt = r2_score(data['Churn'][X_test.index], y_pred_dt)

# Accuracy for k-Nearest Neighbors
y_pred_knn = knn.predict(X_test)
accuracy_knn = accuracy_score(data['Churn'][X_test.index], y_pred_knn)
r2_knn = r2_score(data['Churn'][X_test.index], y_pred_knn)

# Accuracy for Naive Bayes
y_pred_nb = nb.predict(X_test)
accuracy_nb = accuracy_score(data['Churn'][X_test.index], y_pred_nb)
r2_nb = r2_score(data['Churn'][X_test.index], y_pred_nb)

# Print Results
print("\nClassification Model Performance:")
print("Logistic Regression: Accuracy =", accuracy_lr)
print("Decision Tree: Accuracy =", accuracy_dt)
print("k-Nearest Neighbors: Accuracy =", accuracy_knn)
print("Naive Bayes: Accuracy =", accuracy_nb)
```

Figure 26: comparing performance code

```
Classification Model Performance:
Logistic Regression: Accuracy = 0.9121693121693122
Decision Tree: Accuracy = 0.9343915343915344
k-Nearest Neighbors: Accuracy = 0.8243386243386244
Naive Bayes: Accuracy = 0.817989417989418
```

Figure 27: performance between classification models

ROC/AUC

```
# Plot ROC curves for classification models
plt.figure(figsize=(10, 8))

# Logistic Regression
fpr_lr, tpr_lr, _ = roc_curve(data['Churn'][X_test.index], y_prob_lr)
plt.plot( *args: fpr_lr, tpr_lr, label="Logistic Regression")

# Decision Tree
fpr_dt, tpr_dt, _ = roc_curve(data['Churn'][X_test.index], y_prob_dt)
plt.plot( *args: fpr_dt, tpr_dt, label="Decision Tree")

# k-Nearest Neighbors
fpr_knn, tpr_knn, _ = roc_curve(data['Churn'][X_test.index], y_prob_knn)
plt.plot( *args: fpr_knn, tpr_knn, label="k-Nearest Neighbors")

# Naive Bayes
fpr_nb, tpr_nb, _ = roc_curve(data['Churn'][X_test.index], y_prob_nb)
plt.plot( *args: fpr_nb, tpr_nb, label="Naive Bayes")

plt.title("ROC Curves for Classification Models")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend()
plt.grid()
plt.show()
```

Figure 28:ROC/AUC code

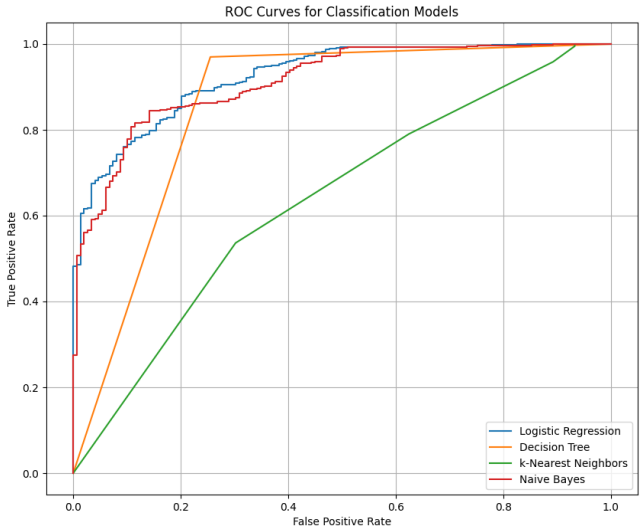

Figure 29: ROC/AUC figure

Confusion Matrix

Naive Bayes

```
# Confusion Matrix for Naive Bayes
cm_nb = confusion_matrix(data['Churn'][X_test.index], y_pred_nb)
disp_nb = ConfusionMatrixDisplay(confusion_matrix=cm_nb, display_labels=["No Churn", "Churn"])
disp_nb.plot(cmap='Purples')
plt.title("Confusion Matrix - Naive Bayes")
plt.show()
```

Figure 30: Confusion matrix Naive Bayes code

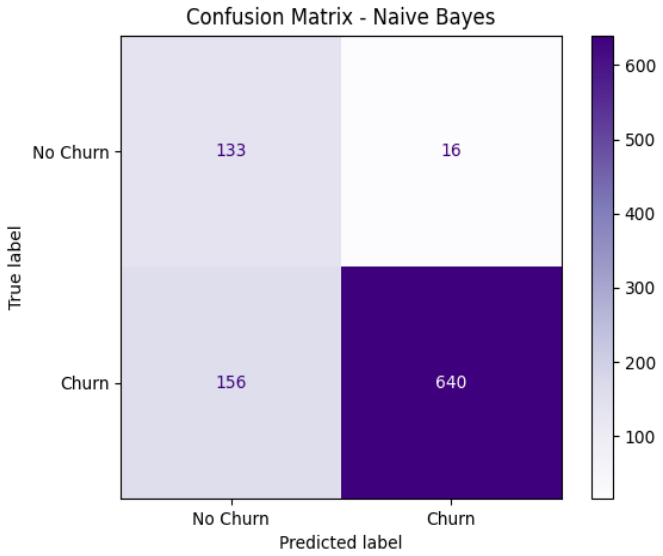

Figure 31: Confusion matrix Naive Bayes

Decision Tree

```
# Confusion Matrix for Decision Tree
cm_dt = confusion_matrix(data['Churn'][X_test.index], y_pred_dt)
disp_dt = ConfusionMatrixDisplay(confusion_matrix=cm_dt, display_labels=["No Churn", "Churn"])
disp_dt.plot(cmap='Greens')
plt.title("Confusion Matrix - Decision Tree")
plt.show()
```

Figure 32: decision tree matrix code

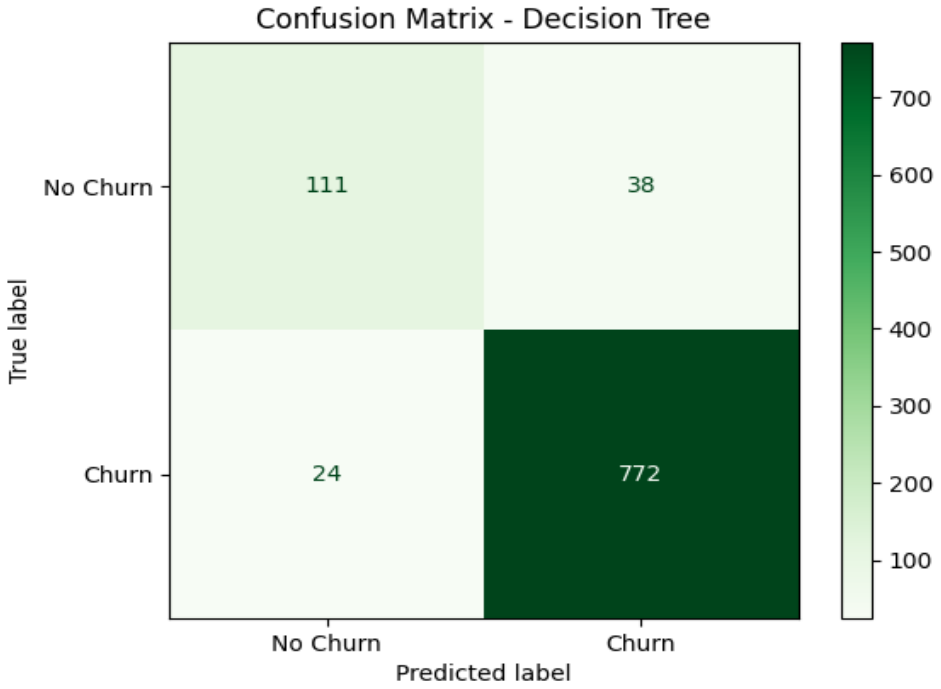

Figure 33:Decision tree matrix

KNN

```
# Confusion Matrix for k-Nearest Neighbors
cm_knn = confusion_matrix(data['Churn'][X_test.index], y_pred_knn)
disp_knn = ConfusionMatrixDisplay(confusion_matrix=cm_knn, display_labels=["No Churn", "Churn"])
disp_knn.plot(cmap='Oranges')
plt.title("Confusion Matrix - k-Nearest Neighbors")
plt.show()
```

Figure 34:KNN confusion matrix code

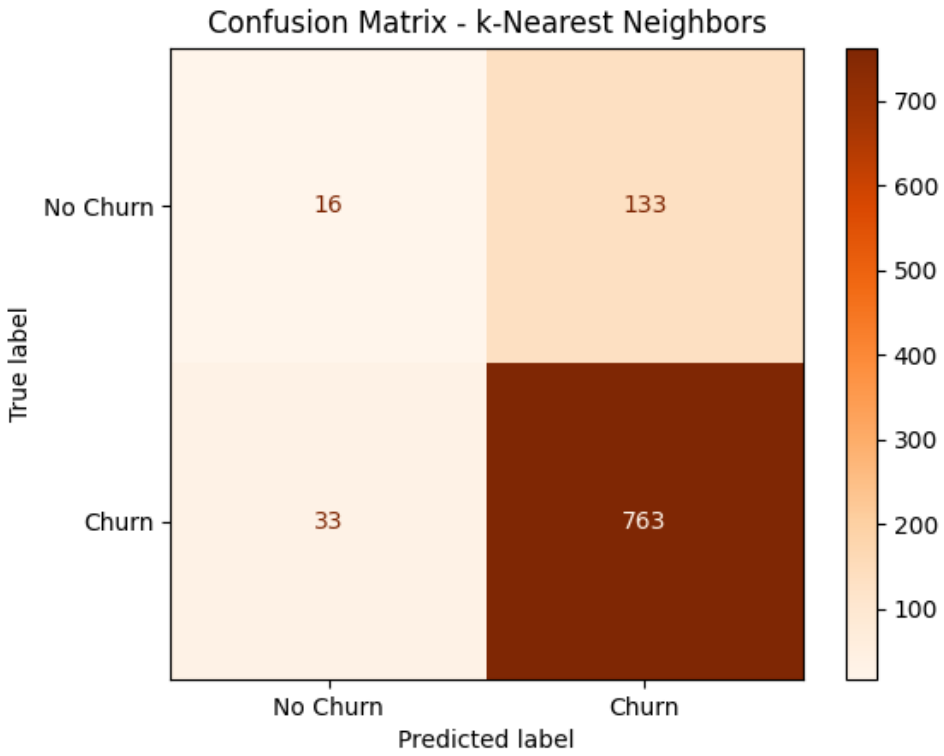

Figure 35: KNN confusion matrix

Logistic Regression

```
# Confusion Matrix for Logistic Regression
cm_lr = confusion_matrix(data['Churn'][X_test.index], y_pred_lr)
disp_lr = ConfusionMatrixDisplay(confusion_matrix=cm_lr, display_labels=["No Churn", "Churn"])
disp_lr.plot(cmap='Blues')
plt.title("Confusion Matrix - Logistic Regression")
plt.show()
```

Figure 36: Logistic regression confusion matrix code

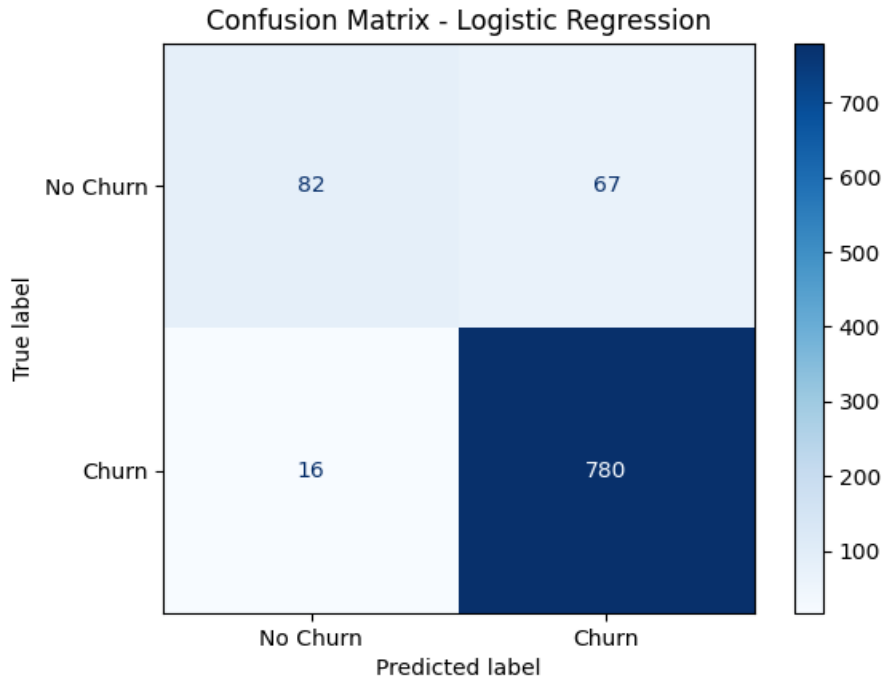

Figure 37:Logistic regression Confusion matrix

Model	Accuracy	Comments
Logistic Regression	0.912	Consistently high accuracy with good generalization capabilities.
Decision Tree	0.934	The highest accuracy but could be prone to overfitting.
k-Nearest Neighbors	0.824	Moderate performance; sensitive to noisy data and scaling.
Naive Bayes	0.818	Lowest accuracy; limited by its assumption of feature independence.

Decision Tree (Accuracy = 0.934):

- **outperforms:**
  - Decision trees can capture complex patterns in the data, leading to the highest accuracy.
  - It handles categorical and numerical features well and does not require feature scaling.

While the high accuracy indicates strong performance, Decision Trees are prone to overfitting, especially if the tree grows too deep. This can cause poor generalization to unseen data unless techniques such as pruning, cross-validation, or ensemble methods (e.g., Random Forest) are applied.

Logistic Regression (Accuracy = 0.912):

Logistic Regression provides robust performance when the dataset has a linear relationship between features and the target variable. The use of feature scaling (via StandardScaler) further improves its efficiency and accuracy.

It is less complex and less prone to overfitting than a Decision Tree. However, it struggles to capture non-linear relationships, which can limit its applicability to datasets with intricate patterns.

k-Nearest Neighbors (Accuracy = 0.824):

kNN reliance on the proximity of data points makes it sensitive to noise and scaling. Additionally, its performance heavily depends on the choice of hyperparameters (k) and can degrade for larger datasets or those with overlapping classes.

kNN works well for smaller, well-separated datasets but is less effective for datasets with complex or noisy distributions.

Naive Bayes (Accuracy = 0.818):

Naive Bayes assumes independence between features, which is rarely the case in real-world datasets. This oversimplification can lead to suboptimal performance when feature dependencies exist.

It is computationally efficient and suitable for high-dimensional datasets. Despite its limitations, it can serve as a baseline model for comparison.