```
!pip install catboost
```

```
Collecting catboost
  Downloading catboost-1.2.8-cp311-cp311-manylinux2014_x86_64.whl.metadata (1.2 kB)
Requirement already satisfied: graphviz in /usr/local/lib/python3.11/dist-packages (fr
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-packages (
Requirement already satisfied: numpy<3.0,>=1.16.0 in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: pandas>=0.24 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: plotly in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: six in /usr/local/lib/python3.11/dist-packages (from ca
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dis
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packa
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.11/dist-packa
Downloading catboost-1.2.8-cp311-cp311-manylinux2014_x86_64.whl (99.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 99.2/99.2 MB 8.0 MB/s eta 0:00:00
Installing collected packages: catboost
Successfully installed catboost-1.2.8
```

```
from google.colab import files
upload = files.upload()
print(upload)
```

```
Choose Files No file chosen  Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving WA_Fn-UseC_-Telco-Customer-Churn.csv to WA_Fn-UseC_-Telco-Customer-Churn.csv
{'WA_Fn-UseC_-Telco-Customer-Churn.csv': b'customerID,gender,SeniorCitizen,Partner,Dep
```

```
import pandas as pd
import numpy as np
import missingno as msno
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import warnings
warnings.filterwarnings('ignore')
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
```

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from xgboost import XGBClassifier
from catboost import CatBoostClassifier
from sklearn import metrics
from sklearn.metrics import roc_curve
from sklearn.metrics import recall_score, confusion_matrix, precision_score, f1_score, acc


df = pd.read_csv('WA_Fn-UseC_-Telco-Customer-Churn.csv')
df.head()

df.shape
df.info()
df.columns.values
df.dtypes
   #   missing values
#   msno.matrix(df);

# Data Manipulation
df['TotalCharges'] = pd.to_numeric(df.TotalCharges, errors='coerce')
df.isnull().sum()
df[np.isnan(df['TotalCharges'])]
df[df['tenure'] == 0].index
df.drop(labels=df[df['tenure'] == 0].index, axis=0, inplace=True)
df[df['tenure'] == 0].index
df.fillna(df["TotalCharges"].mean(), inplace=True)
df.isnull().sum()
df["SeniorCitizen"]= df["SeniorCitizen"].map({0: "No", 1: "Yes"})
df.head()
df["InternetService"].describe(include=['object', 'bool'])
numerical_cols = ['tenure', 'MonthlyCharges', 'TotalCharges']
df[numerical_cols].describe()


# Data Visualization

df["Churn"][df["Churn"]=="No"].groupby(by=df["gender"]).count()
df["Churn"][df["Churn"]=="Yes"].groupby(by=df["gender"]).count()
fig = px.histogram(df, x="Churn", color="Contract", barmode="group", title="<b>Customer co
fig.update_layout(width=700, height=500, bargap=0.1)
fig.show()
fig = px.histogram(df, x="Churn", color="PaymentMethod", title="<b>Customer Payment Method
fig.update_layout(width=700, height=500, bargap=0.1)
fig.show()
df["InternetService"].unique()
df[df["gender"]=="Male"][["InternetService", "Churn"]].value_counts()
df[df["gender"]=="Female"][["InternetService", "Churn"]].value_counts()
fig = go.Figure()

fig.add_trace(go.Bar(
  x = [['Churn:No', 'Churn:No', 'Churn:Yes', 'Churn:Yes'],
```

```
        ["Female", "Male", "Female", "Male"]],
  y = [965, 992, 219, 240],
  name = 'DSL',
))

fig.add_trace(go.Bar(
  x = [['Churn:No', 'Churn:No', 'Churn:Yes', 'Churn:Yes'],
       ["Female", "Male", "Female", "Male"]],
  y = [889, 910, 664, 633],
  name = 'Fiber optic',
))
fig.add_trace(go.Bar(
  x = [['Churn:No', 'Churn:No', 'Churn:Yes', 'Churn:Yes'],
       ["Female", "Male", "Female", "Male"]],
  y = [690, 717, 56, 57],
  name = 'No Internet',
))

fig.update_layout(title_text="<b>Churn Distribution w.r.t. Internet Service and Gender</b>

fig.show()

color_map = {"Yes": "#FF97FF", "No": "#AB63FA"}
fig = px.histogram(df, x="Churn", color="Dependents", barmode="group", title="<b>Dependent
fig.update_layout(width=700, height=500, bargap=0.1)
fig.show()
color_map = {"Yes": '#FFA15A', "No": '#00CC96'}
fig = px.histogram(df, x="Churn", color="Partner", barmode="group", title="<b>Chrun distri
fig.update_layout(width=700, height=500, bargap=0.1)
fig.show()
color_map = {"Yes": '#00CC96', "No": '#B6E880'}
fig = px.histogram(df, x="Churn", color="SeniorCitizen", title="<b>Chrun distribution w.r.
fig.update_layout(width=700, height=500, bargap=0.1)
fig.show()

color_map = {"Yes": "#FF97FF", "No": "#AB63FA"}
fig = px.histogram(df, x="Churn", color="OnlineSecurity", barmode="group", title="<b>Churn
fig.update_layout(width=700, height=500, bargap=0.1)
fig.show()
color_map = {"Yes": '#FFA15A', "No": '#00CC96'}
fig = px.histogram(df, x="Churn", color="PaperlessBilling",  title="<b>Chrun distribution
fig.update_layout(width=700, height=500, bargap=0.1)
fig.show()
fig = px.histogram(df, x="Churn", color="TechSupport",barmode="group",  title="<b>Chrun di
fig.update_layout(width=700, height=500, bargap=0.1)
fig.show()

color_map = {"Yes": '#00CC96', "No": '#B6E880'}
fig = px.histogram(df, x="Churn", color="PhoneService", title="<b>Chrun distribution w.r.t
fig.update_layout(width=700, height=500, bargap=0.1)
fig.show()

sns.set_context("paper",font_scale=1.1)
ax = sns.kdeplot(df.MonthlyCharges[(df["Churn"] == 'No') ],
                 color="Red", shade = True);
ax = sns.kdeplot(df.MonthlyCharges[(df["Churn"] == 'Yes') ],
```

```
                ax =ax, color="Blue", shade= True);
ax.legend(["Not Churn","Churn"],loc='upper right');
ax.set_ylabel('Density');
ax.set_xlabel('Monthly Charges');
ax.set_title('Distribution of monthly charges by churn');

ax = sns.kdeplot(df.TotalCharges[(df["Churn"] == 'No') ],
                color="Gold", shade = True);
ax = sns.kdeplot(df.TotalCharges[(df["Churn"] == 'Yes') ],
                ax =ax, color="Green", shade= True);
ax.legend(["Not Chu0rn","Churn"],loc='upper right');
ax.set_ylabel('Density');
ax.set_xlabel('Total Charges');
ax.set_title('Distribution of total charges by churn');
fig = px.box(df, x='Churn', y = 'tenure')

# Update yaxis properties
fig.update_yaxes(title_text='Tenure (Months)', row=1, col=1)
# Update xaxis properties
fig.update_xaxes(title_text='Churn', row=1, col=1)

# Update size and title
fig.update_layout(autosize=True, width=750, height=600,
    title_font=dict(size=25, family='Courier'),
    title='<b>Tenure vs Churn</b>',
)
fig.show()

plt.figure(figsize=(25, 10))

corr = df.apply(lambda x: pd.factorize(x)[0]).corr()

mask = np.triu(np.ones_like(corr, dtype=bool))

ax = sns.heatmap(corr, mask=mask, xticklabels=corr.columns, yticklabels=corr.columns, anno

# Data preprocessing
def object_to_int(dataframe_series):
    if isinstance(dataframe_series, pd.Series) and dataframe_series.dtype == 'object':
        return LabelEncoder().fit_transform(dataframe_series.astype(str))
    return dataframe_series

df = df.apply(lambda x: object_to_int(x))
df.head()

plt.figure(figsize=(14,7))
# df.corr()['Churn'].sort_values(ascending = False)
X = df.drop(columns = ['Churn'])
y = df['Churn'].values
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.30, random_state = 4
def distplot(feature, frame, color='r'):
    plt.figure(figsize=(8,3))
    plt.title("Distribution for {}".format(feature))
    ax = sns.distplot(frame[feature], color= color)

num_cols = ["tenure", 'MonthlyCharges', 'TotalCharges']
```

```
for feat in num_cols: distplot(feat, df)
df_std = pd.DataFrame(StandardScaler().fit_transform(df[num_cols].astype('float64')),
                      columns=num_cols)
for feat in numerical_cols: distplot(feat, df_std, color='c')
cat_cols_ohe =['PaymentMethod', 'Contract', 'InternetService'] # those that need one-hot e
cat_cols_le = list(set(X_train.columns)- set(num_cols) - set(cat_cols_ohe)) #those that ne
scaler= StandardScaler()

X_train[num_cols] = scaler.fit_transform(X_train[num_cols])
X_test[num_cols] = scaler.transform(X_test[num_cols])

# Machine Learning Model Evaluations
# KNN MODEL

knn_model = KNeighborsClassifier(n_neighbors = 11)
knn_model.fit(X_train,y_train)
predicted_y = knn_model.predict(X_test)
accuracy_knn = knn_model.score(X_test,y_test)
print("KNN accuracy:",accuracy_knn)
print(classification_report(y_test, predicted_y))

# SVC
svc_model = SVC(random_state = 1)
svc_model.fit(X_train,y_train)
predict_y = svc_model.predict(X_test)
accuracy_svc = svc_model.score(X_test,y_test)
print("SVM accuracy is :",accuracy_svc)

print(classification_report(y_test, predict_y))


# Random forest
model_rf = RandomForestClassifier(n_estimators=500 , oob_score = True, n_jobs = -1, random
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train the model
model_rf = RandomForestClassifier(n_estimators=100, random_state=42)
model_rf.fit(X_train, y_train)

# Make predictions
prediction_test = model_rf.predict(X_test)
prediction_train = model_rf.predict(X_train)

# Calculate accuracy
accuracy_test = accuracy_score(y_test, prediction_test)
accuracy_train = accuracy_score(y_train, prediction_train)

print("Test Accuracy:", accuracy_test)
print("Train Accuracy:", accuracy_train)

print(classification_report(y_test, prediction_test))
# accuracy = (cm.diagonal().sum() / cm.sum())
# print(f"Accuracy: {accuracy:.2f}")

plt.figure(figsize=(4,3))
sns.heatmap(confusion_matrix(y_test, prediction_test),
```

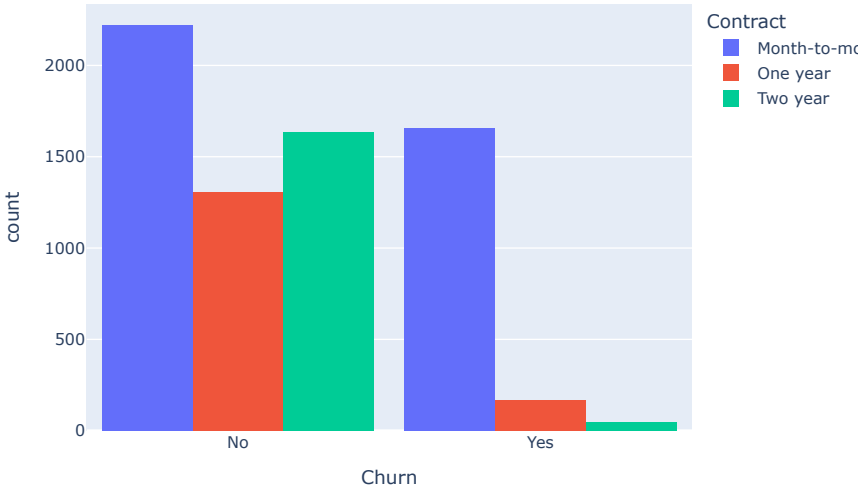
```
                annot=True,fmt = "d",linecolor="k",linewidths=3)

plt.title(" RANDOM FOREST CONFUSION MATRIX",fontsize=14)
plt.show()

y_rfpred_prob = model_rf.predict_proba(X_test)[:,1]
fpr_rf, tpr_rf, thresholds = roc_curve(y_test, y_rfpred_prob)
plt.plot([0, 1], [0, 1], 'k--' )
plt.plot(fpr_rf, tpr_rf, label='Random Forest',color = "r")
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Random Forest ROC Curve',fontsize=16)
plt.show()


 # Logistic Regression
lr_model = LogisticRegression()
lr_model.fit(X_train,y_train)
accuracy_lr = lr_model.score(X_test,y_test)
print("Logistic Regression accuracy is :",accuracy_lr)
lr_pred= lr_model.predict(X_test)
report = classification_report(y_test,lr_pred)
print(report)

plt.figure(figsize=(4,3))
sns.heatmap(confusion_matrix(y_test, lr_pred),
                annot=True,fmt = "d",linecolor="k",linewidths=3)

plt.title("LOGISTIC REGRESSION CONFUSION MATRIX",fontsize=14)
plt.show()

y_pred_prob = lr_model.predict_proba(X_test)[:,1]
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
plt.plot([0, 1], [0, 1], 'k--' )
plt.plot(fpr, tpr, label='Logistic Regression',color = "r")
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Logistic Regression ROC Curve',fontsize=16)
plt.show();

# Decision Tree Classifier

dt_model = DecisionTreeClassifier()
dt_model.fit(X_train,y_train)
predictdt_y = dt_model.predict(X_test)
accuracy_dt = dt_model.score(X_test,y_test)
print("Decision Tree accuracy is :",accuracy_dt)
print(classification_report(y_test, predictdt_y))

# AdaBoostClassifier
a_model = AdaBoostClassifier()
a_model.fit(X_train,y_train)
a_preds = a_model.predict(X_test)
print("AdaBoost Classifier accuracy")
metrics.accuracy_score(y_test, a_preds)
print(classification_report(y_test, a_preds))
```

```
plt.figure(figsize=(4,3))
sns.heatmap(confusion_matrix(y_test, a_preds),
                annot=True,fmt = "d",linecolor="k",linewidths=3)

plt.title("AdaBoost Classifier Confusion Matrix",fontsize=14)
plt.show()

# GradientBoostingClassifier
gb = GradientBoostingClassifier()
gb.fit(X_train, y_train)
gb_pred = gb.predict(X_test)
print("Gradient Boosting Classifier", accuracy_score(y_test, gb_pred))
print(classification_report(y_test, gb_pred))

plt.figure(figsize=(4,3))
sns.heatmap(confusion_matrix(y_test, gb_pred),
                annot=True,fmt = "d",linecolor="k",linewidths=3)

plt.title("Gradient Boosting Classifier Confusion Matrix",fontsize=14)
plt.show()


from sklearn.cluster import KMeans, DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_score


X, _ = make_blobs(n_samples=500, centers=3, cluster_std=0.8, random_state=42)
customer_data = pd.DataFrame(X, columns=['Purchase_Amount', 'Purchase_Frequency'])

# Preprocess data: Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(customer_data)

# --- K-Means Clustering for Customer Segmentation ---
def kmeans_segmentation(X_scaled, n_clusters=3):
    # Initialize and fit K-Means
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    kmeans_labels = kmeans.fit_predict(X_scaled)

    # Calculate silhouette score for cluster quality
    sil_score = silhouette_score(X_scaled, kmeans_labels)

    # Plot results
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=kmeans_labels, cmap='viridis', s=50)
    plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
                c='red', marker='x', s=200, linewidths=3, label='Centroids')
    plt.title(f'K-Means Customer Segmentation\nSilhouette Score: {sil_score:.3f}')
    plt.xlabel('Scaled Purchase Amount')
    plt.ylabel('Scaled Purchase Frequency')
    plt.legend()

    return kmeans_labels, sil_score
```

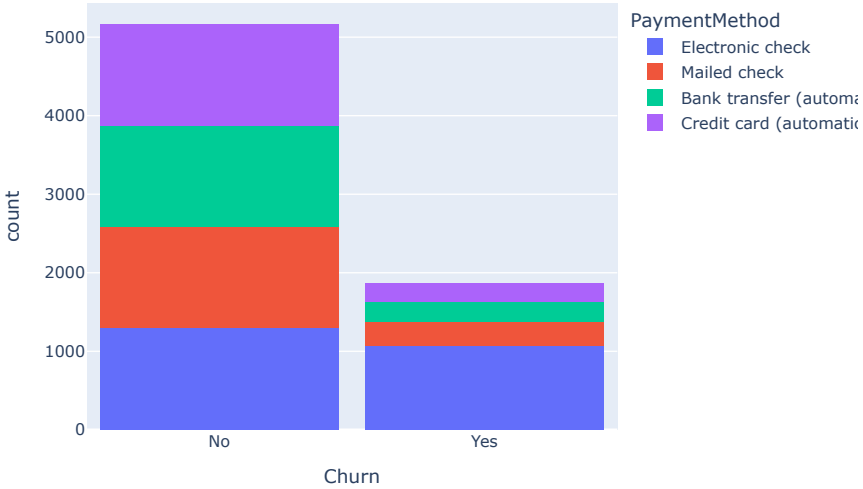
```
# --- DBSCAN for Anomaly Detection in Customers ---
def dbscan_anomaly_detection(X_scaled, eps=0.5, min_samples=5):
    # Initialize and fit DBSCAN
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)
    dbscan_labels = dbscan.fit_predict(X_scaled)

    # Anomalies are labeled as -1
    anomalies = X_scaled[dbscan_labels == -1]
    core_samples = X_scaled[dbscan_labels != -1]

    # Plot results
    plt.subplot(1, 2, 2)
    plt.scatter(core_samples[:, 0], core_samples[:, 1], c=dbscan_labels[dbscan_labels != -
                cmap='viridis', s=50, label='Core Customers')
    plt.scatter(anomalies[:, 0], anomalies[:, 1], c='red', marker='x', s=100,
                label='Anomalous Customers')
    plt.title('DBSCAN Anomaly Detection')
    plt.xlabel('Scaled Purchase Amount')
    plt.ylabel('Scaled Purchase Frequency')
    plt.legend()

    return dbscan_labels, anomalies

# Run both algorithms
kmeans_labels, sil_score = kmeans_segmentation(X_scaled)
dbscan_labels, anomalies = dbscan_anomaly_detection(X_scaled)

# Display the plots
plt.tight_layout()
plt.show()

# Print results
print(f"K-Means Silhouette Score: {sil_score:.3f}")
print(f"K-Means Cluster Sizes: {np.bincount(kmeans_labels)}")
print(f"Number of Anomalous Customers (DBSCAN): {len(anomalies)}")

# missing values

msno.matrix(df);
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7043 entries, 0 to 7042
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   customerID        7043 non-null   object 
 1   gender            7043 non-null   object 
 2   SeniorCitizen     7043 non-null   int64  
 3   Partner           7043 non-null   object 
 4   Dependents        7043 non-null   object 
 5   tenure            7043 non-null   int64  
 6   PhoneService      7043 non-null   object 
 7   MultipleLines     7043 non-null   object 
 8   InternetService   7043 non-null   object 
 9   OnlineSecurity    7043 non-null   object 
 10  OnlineBackup      7043 non-null   object 
 11  DeviceProtection  7043 non-null   object 
 12  TechSupport       7043 non-null   object 
 13  StreamingTV       7043 non-null   object 
 14  StreamingMovies   7043 non-null   object 
 15  Contract          7043 non-null   object 
 16  PaperlessBilling  7043 non-null   object 
 17  PaymentMethod     7043 non-null   object 
 18  MonthlyCharges    7043 non-null   float64
 19  TotalCharges      7043 non-null   object 
 20  Churn             7043 non-null   object 
dtypes: float64(1), int64(2), object(18)
memory usage: 1.1+ MB
```

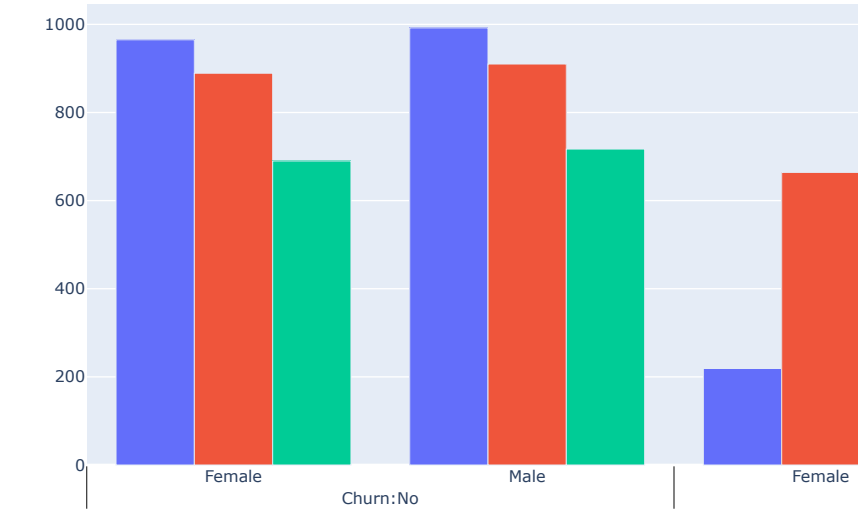
Customer contract distribution



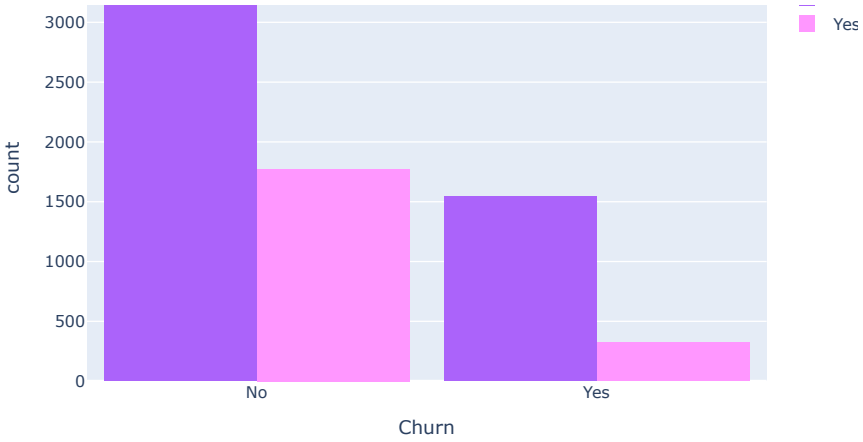
Customer Payment Method distribution w.r.t. Churn



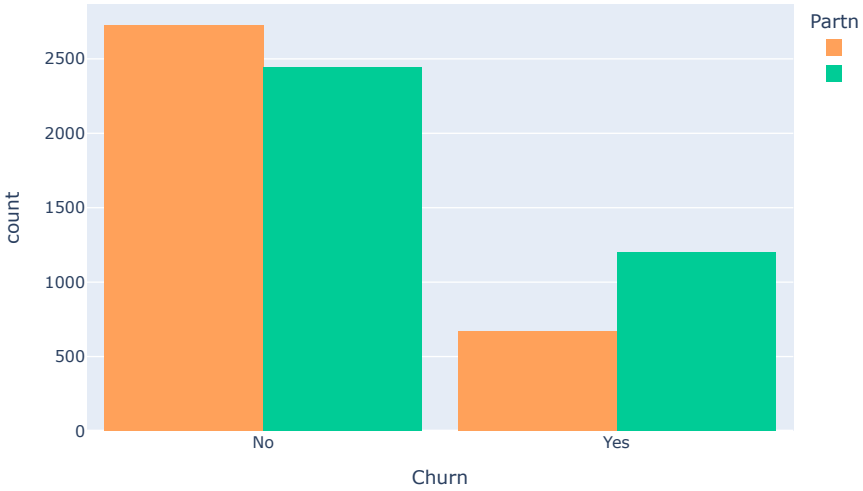
Churn Distribution w.r.t. Internet Service and Gender



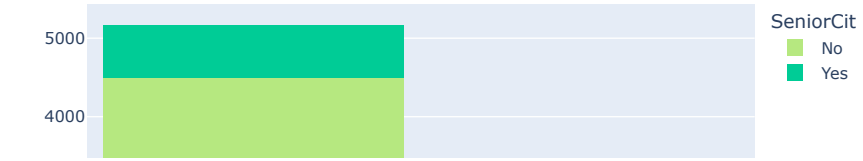
Dependents distribution

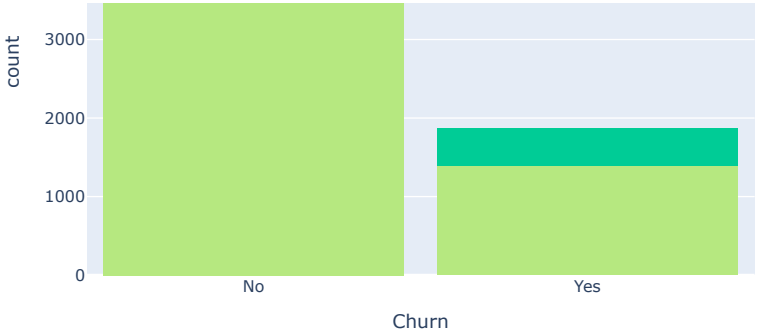


Chrun distribution w.r.t. Partners

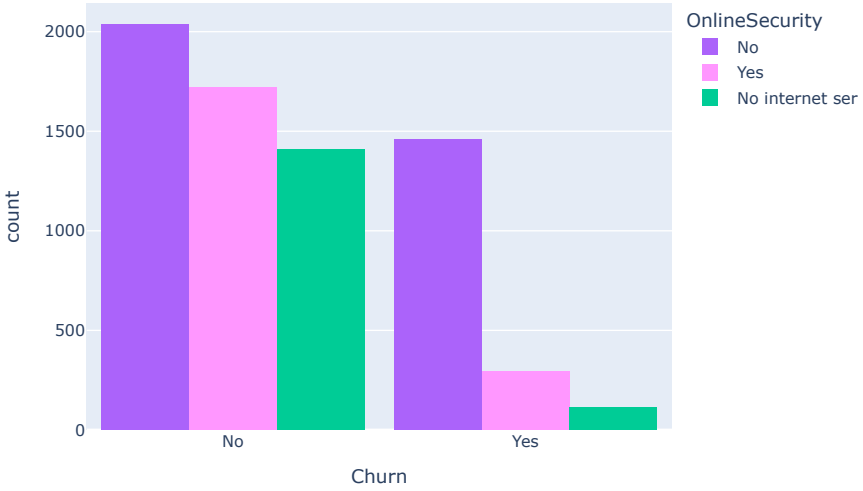


Chrun distribution w.r.t. Senior Citizen

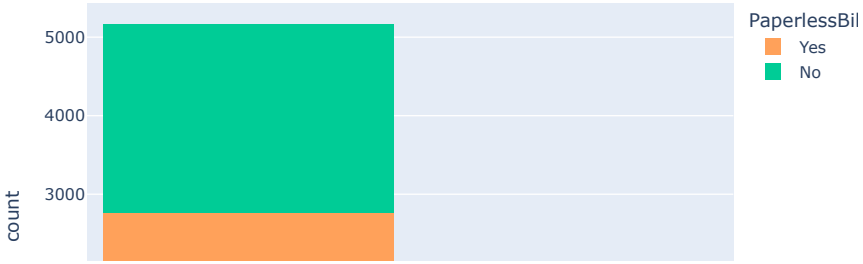


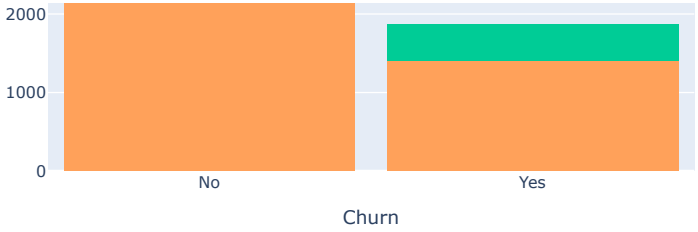


Churn w.r.t Online Security

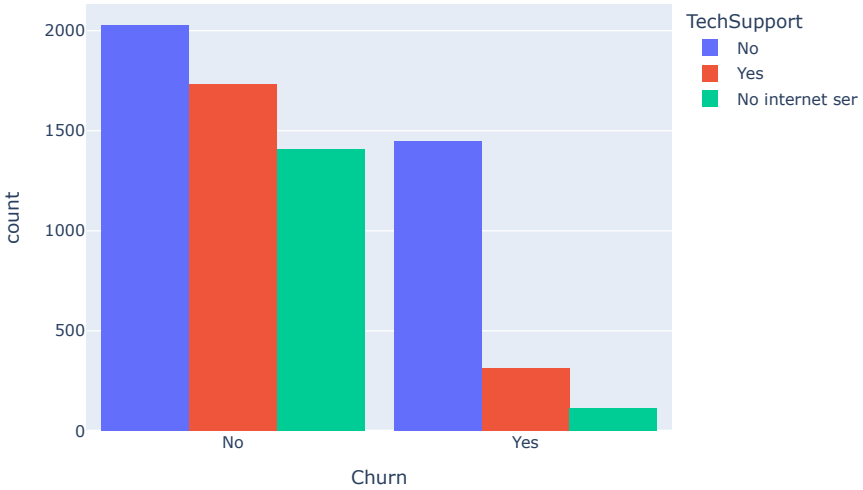


Chrun distribution w.r.t. Paperless Billing

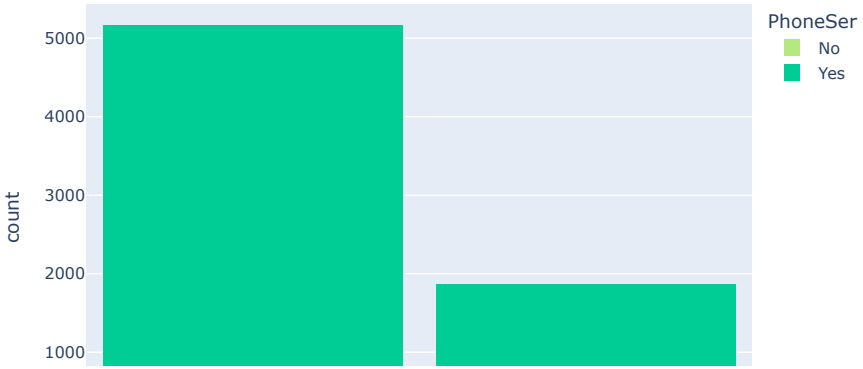


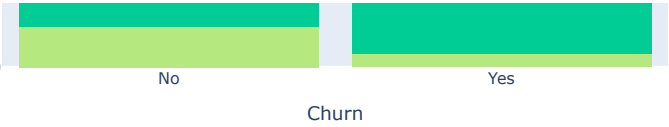


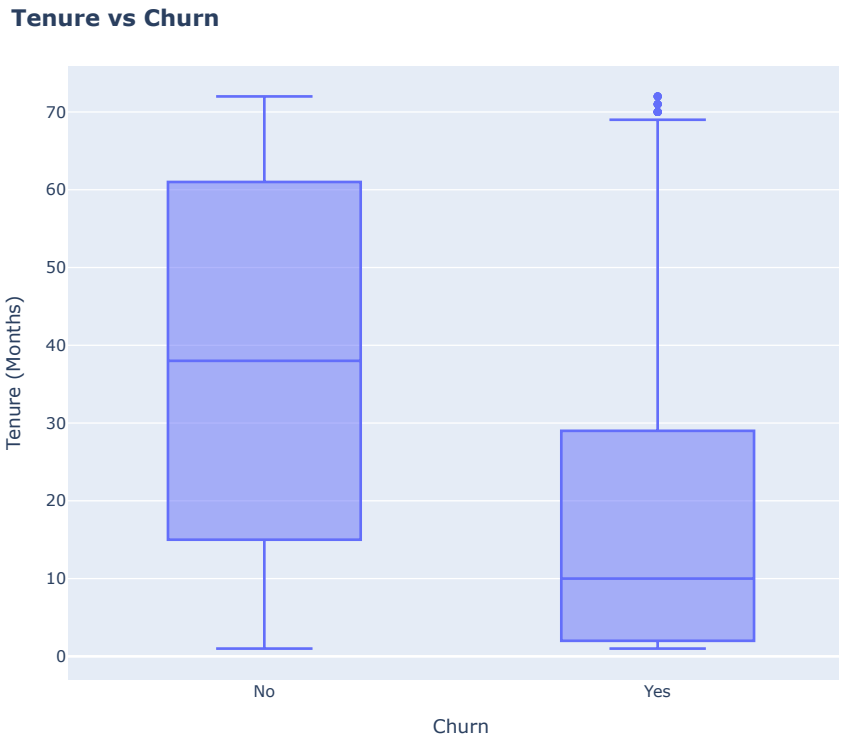
Chrun distribution w.r.t. TechSupport



Chrun distribution w.r.t. Phone Service

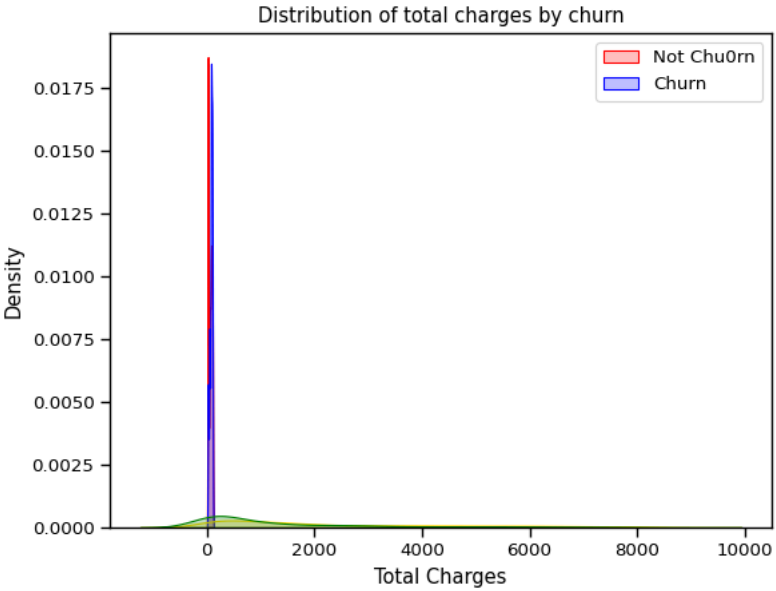


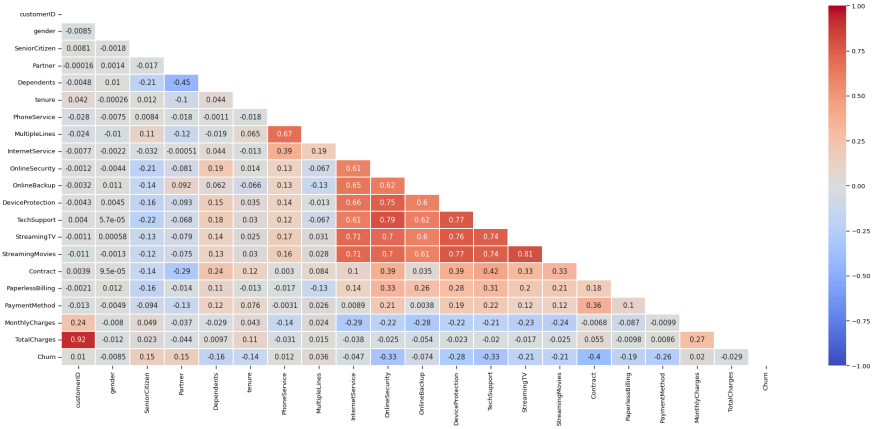




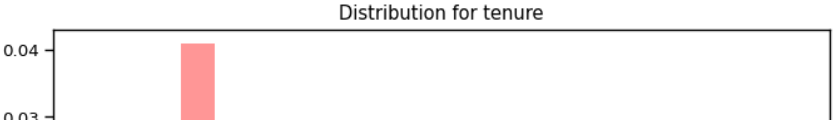

```
KNN accuracy: 0.723696682464455
              precision    recall  f1-score   support

           0       0.74      0.96      0.84      1549
           1       0.39      0.07      0.12       561

    accuracy                           0.72      2110
   macro avg       0.56      0.51      0.48      2110
weighted avg       0.65      0.72      0.64      2110

SVM accuracy is : 0.7341232227488151
              precision    recall  f1-score   support

           0       0.73      1.00      0.85      1549
           1       0.00      0.00      0.00       561

    accuracy                           0.73      2110
   macro avg       0.37      0.50      0.42      2110
weighted avg       0.54      0.73      0.62      2110
```

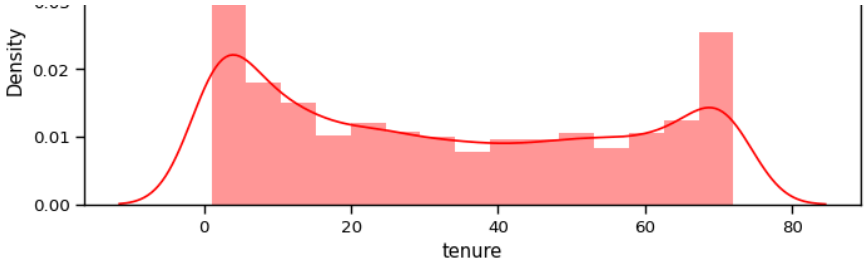
```
Test Accuracy: 0.7846481876332623
Train Accuracy: 1.0
              precision    recall  f1-score   support

           0       0.82      0.90      0.86      1033
           1       0.63      0.45      0.53       374

    accuracy                           0.78      1407
   macro avg       0.73      0.68      0.69      1407
weighted avg       0.77      0.78      0.77      1407
```

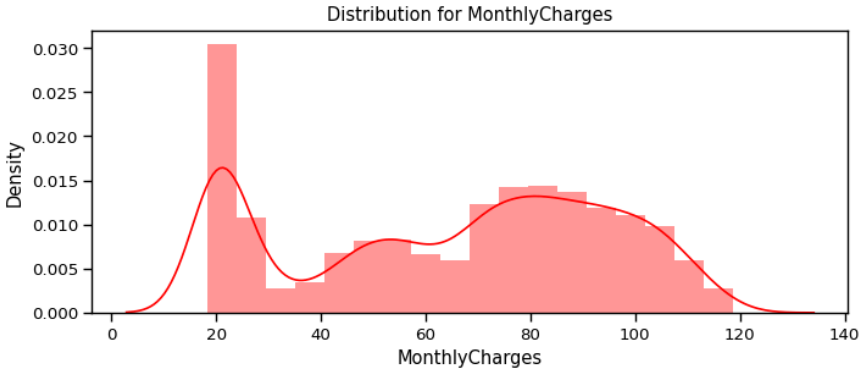


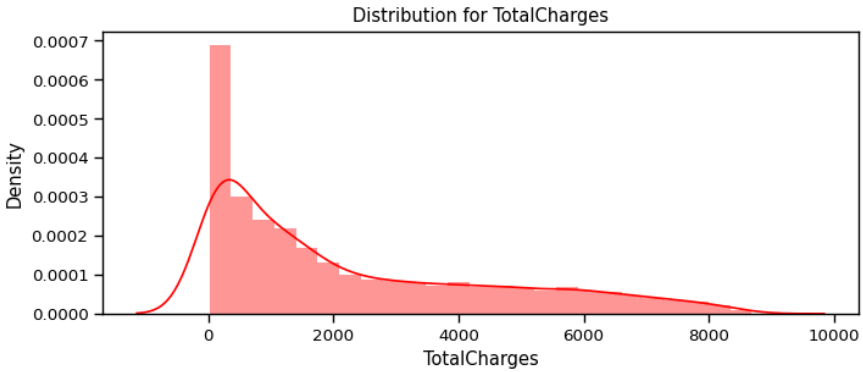


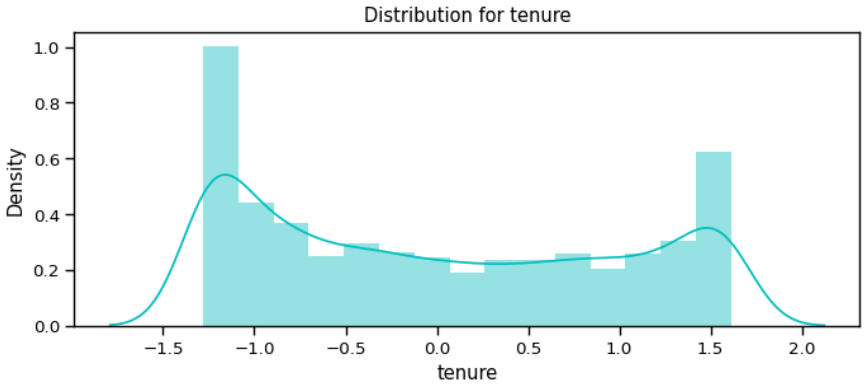
```
<Figure size 1400x700 with 0 Axes>
```

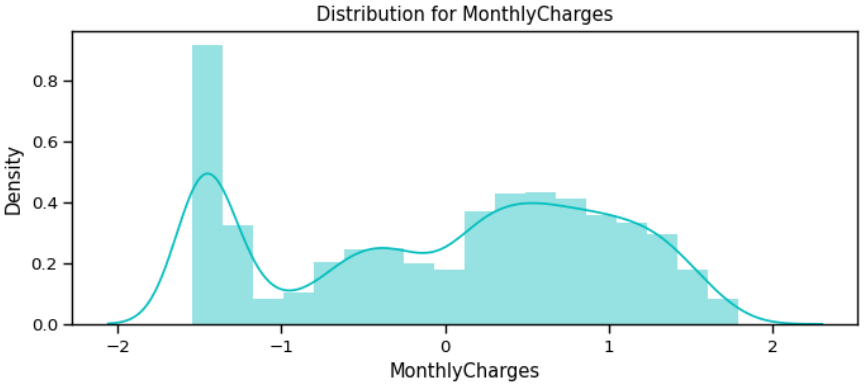


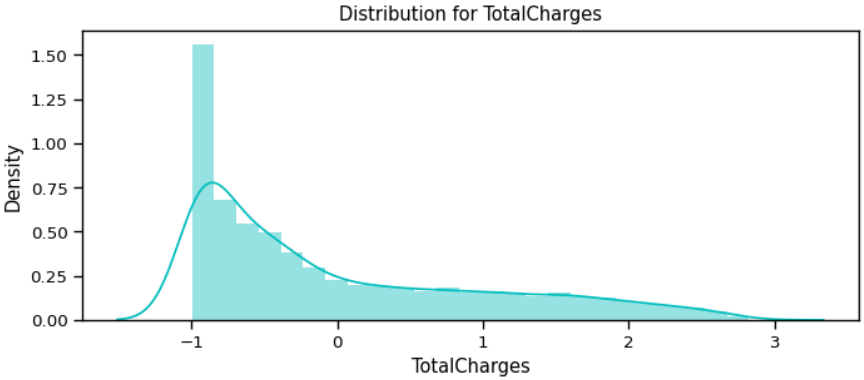

Density
0.02
0.01
0.00
0
20
40
60
80
tenure

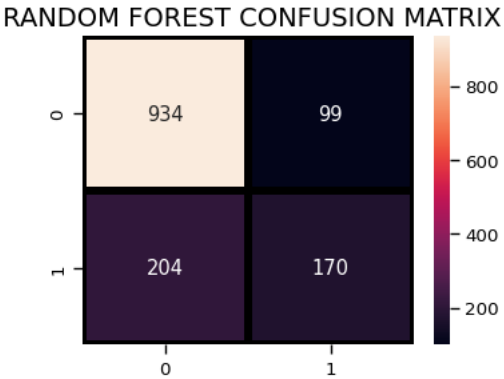

Distribution for MonthlyCharges
Density
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
20
40
60
80
100
120
140
MonthlyCharges

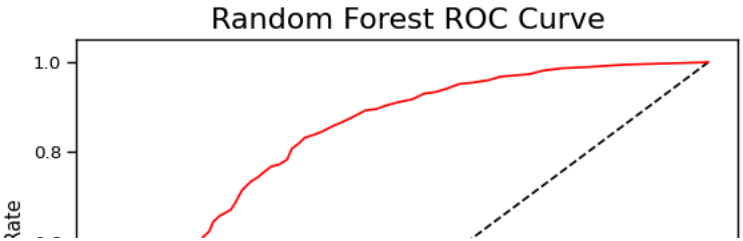

Distribution for TotalCharges
Density
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
0
2000
4000
6000
8000
10000
TotalCharges

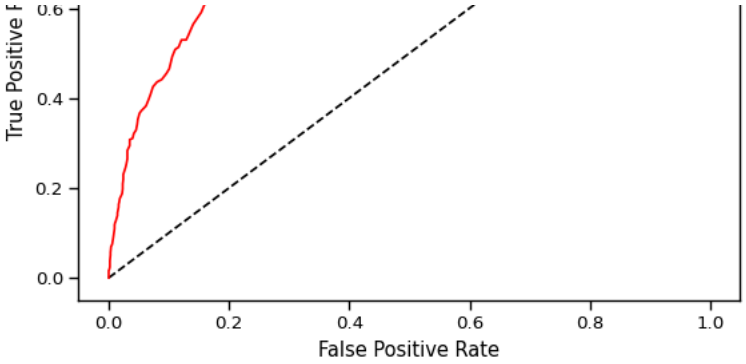

Distribution for tenure
Density
1.0
0.8
0.6
0.4
0.2
0.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
tenure


Distribution for MonthlyCharges
Density
MonthlyCharges
0.0
0.2
0.4
0.6
0.8
−2
−1
0
1
2

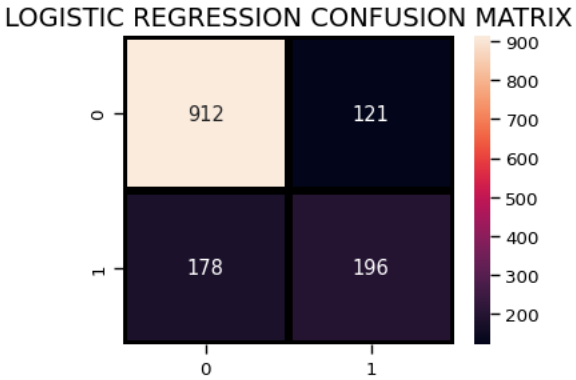

Distribution for TotalCharges
Density
TotalCharges
0.00
0.25
0.50
0.75
1.00
1.25
1.50
−1
0
1
2
3

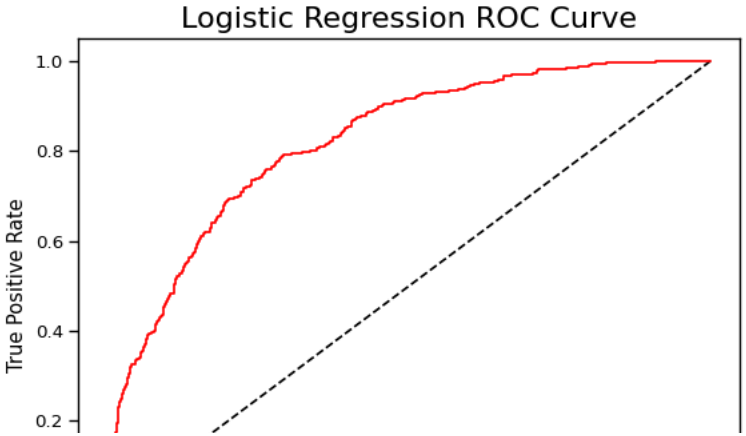

RANDOM FOREST CONFUSION MATRIX
934
99
204
170
0
1
0
1
800
600
400
200

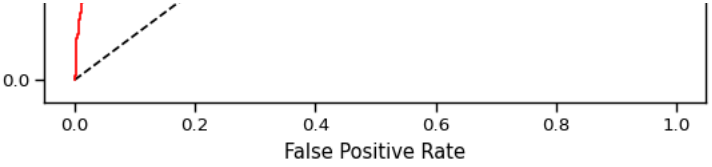

Random Forest ROC Curve
1.0
0.8



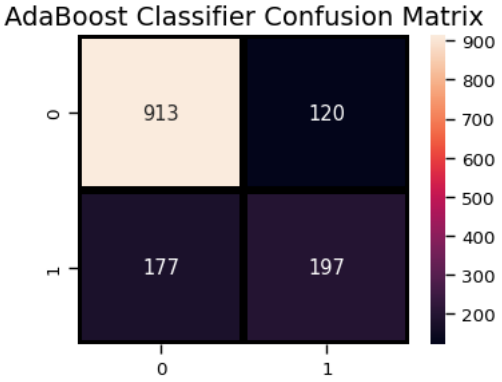

```
Logistic Regression accuracy is : 0.7874911158493249
              precision    recall  f1-score   support

           0       0.84      0.88      0.86      1033
           1       0.62      0.52      0.57       374

    accuracy                           0.79      1407
   macro avg       0.73      0.70      0.71      1407
weighted avg       0.78      0.79      0.78      1407
```



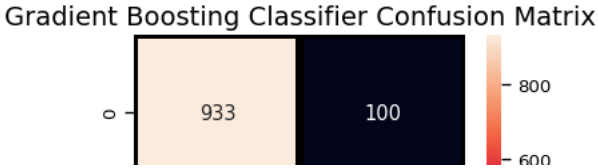


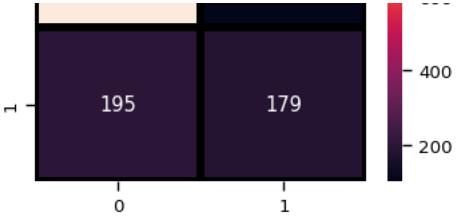


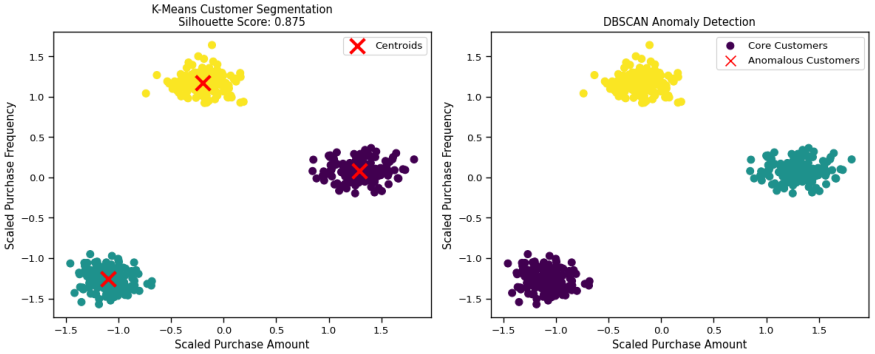

```
Decision Tree accuracy is : 0.7327647476901208
              precision    recall  f1-score   support

           0       0.82      0.81      0.82      1033
           1       0.50      0.53      0.51       374

    accuracy                           0.73      1407
   macro avg       0.66      0.67      0.66      1407
weighted avg       0.74      0.73      0.74      1407

AdaBoost Classifier accuracy
              precision    recall  f1-score   support

           0       0.84      0.88      0.86      1033
           1       0.62      0.53      0.57       374

    accuracy                           0.79      1407
   macro avg       0.73      0.71      0.72      1407
weighted avg       0.78      0.79      0.78      1407
```



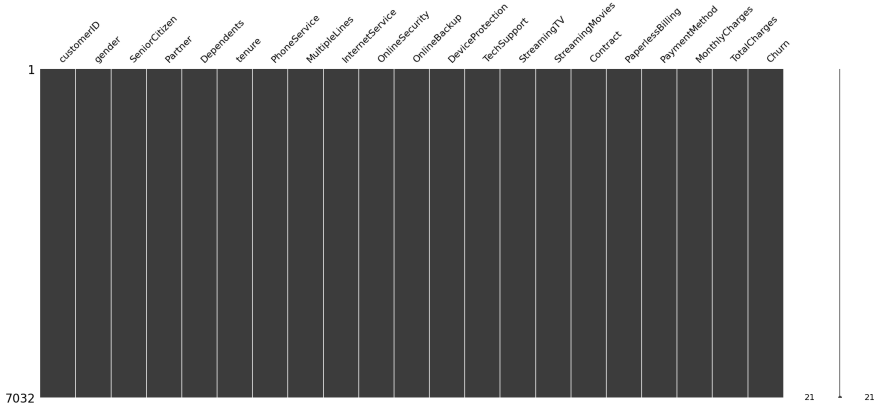

```
Gradient Boosting Classifier 0.7903340440653873
              precision    recall  f1-score   support

           0       0.83      0.90      0.86      1033
           1       0.64      0.48      0.55       374

    accuracy                           0.79      1407
   macro avg       0.73      0.69      0.71      1407
weighted avg       0.78      0.79      0.78      1407
```








```
K-Means Silhouette Score: 0.875
K-Means Cluster Sizes: [167 166 167]
Number of Anomalous Customers (DBSCAN): 0
```



```
import re
import random
import datetime # Import the datetime module for dynamic date and time

# --- 1. Data/Knowledge Base (for a simple rule-based chatbot) ---
# This dictionary represents our chatbot's "knowledge."
# Each key is a pattern (regex), and the value is a list of possible responses.
# In a real data science application, this would be replaced by:
# - A large dataset of conversations (for training)
# - Embeddings for semantic understanding
# - A sophisticated model (e.g., Transformer-based LLM)
chatbot_knowledge = {
    r".*hello|hi|hey.*": [
        "Hello there! How can I help you today?",
        "Hi! What's on your mind?",
        "Hey! Good to see you. Ask me anything.",
    ],
    r".*how are you.*": [
        "I'm just a program, so I don't have feelings, but I'm ready to assist you!",
        "I'm functioning perfectly, thanks for asking! And you?",
        "All good here! How about yourself?",
    ],
    r".*your name.*": [
        "I am a simple AI chatbot, I don't have a name.",
        "You can call me Chatbot.",
        "I'm an AI assistant designed to help you.",
    ],
    r".*what can you do.*": [
        "I can answer basic questions, provide information, or just chat with you.",
        "I'm here to assist with your queries and provide general information.",
        "I can respond to various prompts and engage in simple conversations.",
    ],
    r".*bye|goodbye|see you.*": [
        "Goodbye! Have a great day!",
        "See you later! Feel free to come back if you have more questions.",
        "Farewell! It was nice chatting.",
    ],
    r".*thank you|thanks.*": [
        "You're welcome!",
        "No problem at all!",
        "Glad I could help!",
    ],
    r".*announce|announcement.*": [ # Added pattern for announcements
        "I don't have any specific announcements at the moment, but I'm here to help!",
        "Currently, there are no new announcements. Is there something specific you're look
    ],
    # Default response for when no pattern matches
    r".*": [
        "I'm not sure I understand. Can you rephrase that?",
        "That's an interesting thought, but I don't have an answer for that right now.",
        "Could you please provide more details?",
        "I'm still learning. Could you try asking something else?",
    ]
}

# --- 2. Chatbot Logic ---
```

```
def get_chatbot_response(user_input):
    """
    Determines the chatbot's response based on the user's input.
    For this simple example, it uses pattern matching.
    In a dynamic AI chatbot, this would involve:
    - Text vectorization (e.g., TF-IDF, Word2Vec, BERT embeddings)
    - A machine learning model (e.g., classification, sequence-to-sequence model)
    - Potentially an LLM API call for generative responses.
    """
    user_input = user_input.lower() # Convert to lowercase for easier matching

    # Handle specific dynamic queries first
    if re.search(r".*date.*", user_input):
        today = datetime.date.today()
        return f"Today's date is {today.strftime('%B %d, %Y')}."
    elif re.search(r".*time.*", user_input):
        now = datetime.datetime.now()
        return f"The current time is {now.strftime('%I:%M %p')}."

    # Fallback to general pattern matching
    for pattern, responses in chatbot_knowledge.items():
        if re.search(pattern, user_input):
            # If a pattern matches, pick a random response from the list
            return random.choice(responses)
    return "I'm sorry, I don't have a response for that." # Fallback, though '.*' should ca

# --- 3. Main Chat Loop ---

def start_chatbot():
    """
    Initiates the interactive chatbot session.
    """
    print("Welcome to the simple AI Chatbot! Type 'quit' to exit.")
    while True:
        user_input = input("You: ")
        if user_input.lower() == 'quit':
            print("Chatbot: Goodbye!")
            break

        response = get_chatbot_response(user_input)
        print(f"Chatbot: {response}")

# --- Run the Chatbot ---
if __name__ == "__main__":
    start_chatbot()



import random
response = {
    "hello" :"Hello ! How can help you",
    "Hi":"Hi There!",
    "how are you":"I am fine, Thank you ",
    "bye" :"GoodBye! Have a great day"
}
def chatbot():
```

```
        while True:
            user_input =input("you").lower()
            if user_input=="exit":
                print("chatbot: GoodBye!")
                break
            response =response.get(user_input, "sorry i can't understand")
            print(f"chatbot:{response}")
        # Remove the recursive call to chatbot() to avoid infinite loop
        # chatbot()



    import nltk
    from nltk.sentiment import SentimentIntensityAnalyzer
    import spacy
    import random
    from datetime import datetime

    # --- NLTK Data Download (Run once) ---
    # This block ensures that the necessary NLTK data for sentiment analysis is available.
    # It will download 'vader_lexicon' if it's not already present on your system.
    try:
        nltk.data.find('sentiment/vader_lexicon.zip')
    except LookupError:
        print("Downloading 'vader_lexicon' for NLTK sentiment analysis...")
        nltk.download('vader_lexicon')
        print("Download complete.")



    # --- SpaCy Model Loading (Run once) ---
    # This block loads the 'en_core_web_sm' model for Named Entity Recognition (NER).
    # If the model is not found, it will attempt to download it.
    # Note: For more advanced NLP tasks, you might consider larger SpaCy models (e.g., en_core_
    try:
        # Attempt to load the small English model
        nlp_spacy = spacy.load("en_core_web_sm")
        print("SpaCy 'en_core_web_sm' model loaded successfully.")
    except OSError:
        # If the model is not found, download it
        print("SpaCy 'en_core_web_sm' model not found. Attempting to download...")
        try:
            spacy.cli.download("en_core_web_sm")
            nlp_spacy = spacy.load("en_core_web_sm")
            print("SpaCy 'en_core_web_sm' model downloaded and loaded successfully.")
        except Exception as e:
            print(f"Error downloading or loading SpaCy model: {e}")
            print("Please ensure you have an active internet connection or try running 'python
            nlp_spacy = None # Set to None if loading fails, handle gracefully later


    class DynamicAIChatbot:
        """
        A foundational class for a Dynamic AI Chatbot, demonstrating core NLP,
        context management, and response generation capabilities.
        """
        def __init__(self):
            """
```

```
        Initializes the chatbot with sentiment analyzer, SpaCy NLP model,
        pre-defined intents, responses, and a context storage.
        """
        # Stores conversation context for each user/session.
        # In a real application, this would be persistent (e.g., database).
        self.context = {}

        # Initialize NLTK's VADER sentiment intensity analyzer
        self.sentiment_analyzer = SentimentIntensityAnalyzer()

        # Assign the loaded SpaCy model
        self.nlp_spacy = nlp_spacy

        # Define simple rule-based intents and their associated keywords.
        # In a production system, this would be replaced by a trained
        # machine learning model (e.g., using scikit-learn, TensorFlow, PyTorch).
        self.intents = {
            "greeting": ["hello", "hi", "hey", "greetings", "good morning", "good evening"]
            "farewell": ["bye", "goodbye", "see you", "later", "cya"],
            "query_time": ["time", "what is the time", "current time"],
            "query_date": ["date", "what is the date", "today's date"],
            "thanks": ["thank you", "thanks", "appreciate it"],
            "about_bot": ["who are you", "what can you do", "tell me about yourself"],
            "help": ["help", "can you help me", "i need help"],
            "weather_query": ["weather", "how's the weather", "temperature"] # Example of a
        }

        # Define responses for each intent.
        # Responses can be dynamic (e.g., fetching real-time data) or static.
        self.responses = {
            "greeting": ["Hello there!", "Hi! How can I help you today?", "Hey! Nice to hea
            "farewell": ["Goodbye! Have a great day!", "See you later!", "Bye for now!"],
            "query_time": [f"The current time is {self._get_current_time()}."],
            "query_date": [f"Today's date is {self._get_current_date()}."],
            "thanks": ["You're welcome!", "No problem!", "Glad to help!", "Anytime!"],
            "about_bot": [
                "I am a dynamic AI chatbot designed to assist you with information and conv
                "I can understand your queries, extract important information, and provide
                "Think of me as your digital assistant, ready to chat and help."
            ],
            "help": [
                "I can help with general questions, provide information, and engage in conv
                "What specifically do you need assistance with?",
                "I'm here to assist. How can I be of service?"
            ],
            "weather_query": ["I can tell you the weather, but I'll need to connect to a we
            "no_intent": [
                "I'm not sure I understand. Can you rephrase that?",
                "Could you please provide more details?",
                "I'm still learning. Can you try asking in a different way?",
                "My apologies, I didn't quite catch that. Could you clarify?"
            ]
        }

        # List to store extracted named entities from the last processed message.
        self.named_entities = []
```

```
    def _get_current_time(self):
        """Helper method to get the current local time."""
        return datetime.now().strftime("%I:%M:%S %p") # e.g., 08:48:30 PM

    def _get_current_date(self):
        """Helper method to get the current local date."""
        return datetime.now().strftime("%Y-%m-%d") # e.g., 2025-07-11

    def _recognize_intent(self, text):
        """
        Recognizes the user's intent based on keyword matching.
        This is a simplified approach. In a real-world scenario,
        this would be replaced by a machine learning classifier
        trained on a large dataset of user queries and their corresponding intents.
        """
        text_lower = text.lower()
        for intent, keywords in self.intents.items():
            for keyword in keywords:
                if keyword in text_lower:
                    return intent
        return "no_intent" # Default intent if no match is found

    def _extract_entities(self, text):
        """
        Extracts named entities (e.g., persons, organizations, locations, dates)
        from the input text using SpaCy.
        """
        if not self.nlp_spacy:
            print("SpaCy model not loaded. Cannot perform NER.")
            return []

        doc = self.nlp_spacy(text)
        # Format entities as a list of dictionaries for easier handling
        entities = [{"text": ent.text, "label": ent.label_} for ent in doc.ents]
        self.named_entities = entities # Store for potential use in contextual memory
        return entities

    def _analyze_sentiment(self, text):
        """
        Analyzes the sentiment of the input text using NLTK's VADER.
        Returns 'positive', 'negative', or 'neutral' based on the compound score.
        """
        vs = self.sentiment_analyzer.polarity_scores(text)
        compound_score = vs['compound']

        if compound_score >= 0.05:
            return "positive"
        elif compound_score <= -0.05:
            return "negative"
        else:
            return "neutral"

    def _generate_response(self, intent, sentiment, entities, user_message):
        """
        Generates a response based on the detected intent, sentiment,
        extracted entities, and the user's original message.
```

```
        This function also includes a placeholder for integrating generative AI.
        """
        # Start with a random response template for the detected intent.
        response_template = random.choice(self.responses.get(intent, self.responses["no_int

        # --- Sentiment-based Response Adjustment ---
        # Adjust the response tone or add empathetic remarks based on sentiment.
        if sentiment == "negative" and intent not in ["farewell", "thanks"]:
            response_template += " I sense some negativity. Is there anything specific both
        elif sentiment == "positive" and intent not in ["greeting", "thanks"]:
            response_template += " That's wonderful to hear!"

        # --- Entity-based Response Enhancement ---
        # Incorporate extracted entities into the response for personalization or clarity.
        if entities:
            # Example: If a person's name is mentioned
            person_names = [ent['text'] for ent in entities if ent['label'] == "PERSON"]
            if person_names:
                response_template += f" I noticed you mentioned {', '.join(person_names)}."
            # Example: If a location is mentioned (for weather_query)
            elif intent == "weather_query":
                locations = [ent['text'] for ent in entities if ent['label'] == "GPE" or en
                if locations:
                    response_template = f"Looking up weather for {', '.join(locations)}. Pl
                else:
                    response_template += " Which city are you interested in?"

        # --- Generative AI Integration Placeholder (GPT/Gemini-based models) ---
        # In a real application, if the rule-based system cannot provide a sufficient
        # answer (e.g., for "no_intent" or complex, open-ended queries), you would
        # typically make an API call to a large language model (LLM) like Google's Gemini.
        # This part would usually be handled by the backend server or a client-side
        # JavaScript fetch call in a web application.

        # Example of how you might conceptually integrate a Gemini API call (this is JavaSc
        """
        if intent == "no_intent" or intent == "general_query_requiring_creativity":
            try:
                # This is a conceptual example for a JavaScript fetch call in a web context
                # In Python, you would use a library like 'requests' to call the API.
                #
                # let chatHistory = [];
                # chatHistory.push({ role: "user", parts: [{ text: user_message }] });
                # const payload = { contents: chatHistory };
                # const apiKey = ""; # Your actual API key would be here or loaded from env
                # const apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/g
                #
                # const response = await fetch(apiUrl, {
                #            method: 'POST',
                #            headers: { 'Content-Type': 'application/json' },
                #            body: JSON.stringify(payload)
                #        });
                # const result = response.json();
                # if (result.candidates && result.candidates.length > 0 &&
                #     result.candidates[0].content && result.candidates[0].content.parts &&
                #     result.candidates[0].content.parts.length > 0) {
                #   response_template = result.candidates[0].content.parts[0].text;
```

```
                # } else {
                #   # Handle cases where the response structure is unexpected or content is
                #   print("Gemini API response was empty or malformed.")
                # }
            except Exception as e:
                print(f"Error calling Generative AI API: {e}")
                # Fallback to rule-based or default response if API call fails
        """

        return response_template

    def process_message(self, user_id, message):
        """
        The main entry point for processing a user's message.
        It orchestrates the NLP pipeline, updates conversation context,
        and generates a relevant response.

        Args:
            user_id (str): A unique identifier for the user/session to maintain context.
            message (str): The raw text message from the user.

        Returns:
            str: The chatbot's generated response.
        """
        # Initialize context for the user if it doesn't already exist.
        # This allows the chatbot to remember previous interactions.
        if user_id not in self.context:
            self.context[user_id] = {
                "last_intent": None,
                "last_entities": [],
                "conversation_history": []
            }

        print(f"\n--- Processing message for user '{user_id}': '{message}' ---")

        # 1. Store the user's message in the conversation history for context.
        self.context[user_id]["conversation_history"].append({"user": message})

        # 2. Perform Sentiment Analysis on the user's message.
        sentiment = self._analyze_sentiment(message)
        print(f"  Sentiment Detected: {sentiment}")

        # 3. Recognize the user's intent.
        intent = self._recognize_intent(message)
        print(f"  Intent Recognized: '{intent}'")

        # 4. Extract Named Entities from the message.
        entities = self._extract_entities(message)
        print(f"  Entities Extracted: {entities}")

        # 5. Update the user's context with the latest intent and entities.
        self.context[user_id]["last_intent"] = intent
        self.context[user_id]["last_entities"] = entities

        # 6. Generate the chatbot's response based on the analysis.
        response = self._generate_response(intent, sentiment, entities, message)
        print(f"  Bot Response: '{response}'")
```

```
            print(f"  Bot Response: {response}")

            # 7. Store the chatbot's response in the conversation history.
            self.context[user_id]["conversation_history"].append({"bot": response})

            return response

    # --- Example Usage ---
    # This section demonstrates how to initialize the chatbot and interact with it
    # by simulating multiple user messages from different users.
    if __name__ == "__main__":
        chatbot = DynamicAIChatbot()

        # Simulate interactions for User 1
        user_id_1 = "user_alpha"
        print(f"User '{user_id_1}': Hi there!")
        print(f"Bot: {chatbot.process_message(user_id_1, 'Hi there!')}")

        print(f"\nUser '{user_id_1}': What is the time?")
        print(f"Bot: {chatbot.process_message(user_id_1, 'What is the time?')}")

        print(f"\nUser '{user_id_1}': I am feeling a bit sad today.")
        print(f"Bot: {chatbot.process_message(user_id_1, 'I am feeling a bit sad today.')}")

        print(f"\nUser '{user_id_1}': Thank you for your help.")
        print(f"Bot: {chatbot.process_message(user_id_1, 'Thank you for your help.')}")

        print(f"\nUser '{user_id_1}': Tell me about yourself.")
        print(f"Bot: {chatbot.process_message(user_id_1, 'Tell me about yourself.')}")

        print(f"\nUser '{user_id_1}': What's the weather like in London?")


        # Simulate interactions for User 2 (demonstrates separate context)
        user_id_2 = "user_beta"
        print(f"\nUser '{user_id_2}': Hello, my name is Alice.")
        print(f"Bot: {chatbot.process_message(user_id_2, 'Hello, my name is Alice.')}")

        print(f"\nUser '{user_id_2}': What date is it today?")
        print(f"Bot: {chatbot.process_message(user_id_2, 'What date is it today?')}")

        print(f"\nUser '{user_id_1}': Goodbye.") # User 1 says goodbye
        print(f"Bot: {chatbot.process_message(user_id_1, 'Goodbye.')}")

        print(f"\nUser '{user_id_2}': I want to book a flight to Paris next week.") # Example o
        print(f"Bot: {chatbot.process_message(user_id_2, 'I want to book a flight to Paris next

        # You can inspect the context for each user
        # print("\n--- User Contexts ---")
        # print(f"Context for '{user_id_1}': {chatbot.context.get(user_id_1)}")
        # print(f"Context for '{user_id_2}': {chatbot.context.get(user_id_2)}")
```

```
Welcome to the simple AI Chatbot! Type 'quit' to exit.
You: date
Chatbot: Today's date is July 20, 2025.
You: time
Chatbot: The current time is 04:06 AM.
You: quit
Chatbot: Goodbye!
SpaCy 'en_core_web_sm' model loaded successfully.
User 'user_alpha': Hi there!

--- Processing message for user 'user_alpha': 'Hi there!' ---
  Sentiment Detected: neutral
  Intent Recognized: 'greeting'
  Entities Extracted: []
  Bot Response: 'Hi! How can I help you today?'
Bot: Hi! How can I help you today?

User 'user_alpha': What is the time?

--- Processing message for user 'user_alpha': 'What is the time?' ---
  Sentiment Detected: neutral
  Intent Recognized: 'query_time'
  Entities Extracted: []
  Bot Response: 'The current time is 04:06:51 AM.'
Bot: The current time is 04:06:51 AM.

User 'user_alpha': I am feeling a bit sad today.

--- Processing message for user 'user_alpha': 'I am feeling a bit sad today.' ---
  Sentiment Detected: negative
  Intent Recognized: 'no_intent'
  Entities Extracted: [{'text': 'today', 'label': 'DATE'}]
  Bot Response: 'I'm not sure I understand. Can you rephrase that? I sense some nega
Bot: I'm not sure I understand. Can you rephrase that? I sense some negativity. Is t

User 'user_alpha': Thank you for your help.
```