**Flight Ticket Price Prediction**

**1. Introduction**

The objective of this project is to build a machine learning model to predict TicketPrice (replace this with the actual target column name) based on several features in the dataset. The workflow includes data exploration, preprocessing, feature engineering, dimensionality reduction, model selection, hyperparameter tuning, and evaluation. Multiple models are trained and evaluated, including Linear Regression, Decision Trees, Random Forest, and Gradient Boosting.

**2. Data Exploration**

The first step in any machine learning process is to understand the dataset. We explored the dataset to inspect its structure, understand the types of data we are dealing with, and identify any missing values or inconsistencies.

python

Copy code

print(df.head())

print("\nDataset Info:")

print(df.info())

print("\nSummary statistics:")

print(df.describe())

print("\nMissing values per column:")

print(df.isnull().sum())

**Key Observations:**

* **Data Types**: The dataset consists of both numerical and categorical columns.
* **Missing Values**: There were some missing values in the dataset, which we handled during preprocessing.
* **Statistical Summary**: The describe() function provided insight into the range, mean, and distribution of numerical features.

**3. Data Preprocessing**

**3.1 Handling Missing Values**

To ensure the dataset is complete for model training, we handled missing values. For categorical variables, we filled missing values using the **mode** (most frequent value), and for numerical variables, we filled them with the **mean**.

python

Copy code

for col in df.columns:

if df[col].dtype == 'object':

df[col].fillna(df[col].mode()[0], inplace=True)

else:

df[col].fillna(df[col].mean(), inplace=True)

print("\nMissing values after handling:")

print(df.isnull().sum())

**3.2 One-Hot Encoding**

Categorical features were transformed into numerical values using **One-Hot Encoding**.

python

Copy code

df\_encoded = pd.get\_dummies(df, drop\_first=True)

print("\nData after One-Hot Encoding:")

print(df\_encoded.head())

**4. Feature Scaling**

Feature scaling is essential for algorithms like **PCA** and distance-based algorithms. We applied two scaling methods:

* **StandardScaler** for standardization (mean = 0, std = 1).
* **MinMaxScaler** for normalization (scaled between 0 and 1).

python

Copy code

from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Standardization

scaler\_standard = StandardScaler()

df\_scaled\_standard = df\_encoded.copy()

df\_scaled\_standard[numerical\_columns] = scaler\_standard.fit\_transform(df\_encoded[numerical\_columns])

# Normalization

scaler\_normal = MinMaxScaler()

df\_scaled\_normal = df\_encoded.copy()

df\_scaled\_normal[numerical\_columns] = scaler\_normal.fit\_transform(df\_encoded[numerical\_columns])

**Visualization of Scaled Data (Histograms):**

python

Copy code

df\_scaled\_standard[numerical\_columns].hist(figsize=(12, 10))

plt.show()

df\_scaled\_normal[numerical\_columns].hist(figsize=(12, 10))

plt.show()

**5. Dimensionality Reduction using PCA**

We applied **Principal Component Analysis (PCA)** to reduce the dimensionality of the dataset while retaining 95% of the variance. This helps remove noise and redundant information.

python

Copy code

from sklearn.decomposition import PCA

pca = PCA(n\_components=0.95)

df\_pca = pca.fit\_transform(df\_scaled\_standard[numerical\_columns])

**Explained Variance Ratio Plot:**

python

Copy code

plt.figure(figsize=(8, 6))

plt.bar(range(len(pca.explained\_variance\_ratio\_)), pca.explained\_variance\_ratio\_)

plt.xlabel('PCA Components')

plt.ylabel('Explained Variance Ratio')

plt.title('PCA Explained Variance')

plt.show()

**6. Model Selection**

**6.1 Model Training**

We trained four regression models to predict the target (TicketPrice or actual target column):

* **Linear Regression**
* **Decision Tree Regressor**
* **Random Forest Regressor**
* **Gradient Boosting Regressor**

The dataset was split into training (80%) and testing (20%) sets.

python

Copy code

from sklearn.model\_selection import train\_test\_split

X = df\_encoded.drop(columns=['TicketPrice'])

y = df\_encoded['TicketPrice']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

models = {

'Linear Regression': LinearRegression(),

'Decision Tree': DecisionTreeRegressor(random\_state=42),

'Random Forest': RandomForestRegressor(random\_state=42),

'Gradient Boosting': GradientBoostingRegressor(random\_state=42)

}

for model\_name, model in models.items():

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

print(f'{model\_name} Mean Squared Error: {mse:.4f}')

**7. Hyperparameter Tuning**

We performed hyperparameter tuning to optimize the performance of **Random Forest** and **Gradient Boosting** models using **Grid Search** and **Randomized Search**, respectively.

python

Copy code

# Grid Search for Random Forest

param\_grid\_rf = {

'n\_estimators': [100, 200],

'max\_depth': [10, 20]

}

grid\_search\_rf = GridSearchCV(estimator=random\_forest, param\_grid=param\_grid\_rf, cv=3, n\_jobs=-1)

grid\_search\_rf.fit(X\_train, y\_train)

# Randomized Search for Gradient Boosting

param\_dist\_gb = {

'n\_estimators': [100, 150],

'learning\_rate': [0.05, 0.1],

'max\_depth': [3, 5]

}

random\_search\_gb = RandomizedSearchCV(estimator=gradient\_boosting, param\_distributions=param\_dist\_gb, n\_iter=5, cv=3, n\_jobs=-1)

random\_search\_gb.fit(X\_train, y\_train)

**8. Model Evaluation**

We evaluated each model using key performance metrics such as **Mean Absolute Error (MAE)**, **Mean Squared Error (MSE)**, **Root Mean Squared Error (RMSE)**, and **R-squared (R²)**.

python

Copy code

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

def evaluate\_model(model, X\_test, y\_test):

y\_pred = model.predict(X\_test)

mae = mean\_absolute\_error(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

rmse = np.sqrt(mse)

r2 = r2\_score(y\_test, y\_pred)

print(f"Model: {model.\_\_class\_\_.\_\_name\_\_}")

print(f"MAE: {mae:.4f}, MSE: {mse:.4f}, RMSE: {rmse:.4f}, R²: {r2:.4f}")

return {'Model': model.\_\_class\_\_.\_\_name\_\_, 'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'R²': r2}

evaluation\_results = []

for model\_name, model in models.items():

result = evaluate\_model(model, X\_test, y\_test)

evaluation\_results.append(result)

# Convert to DataFrame

eval\_df = pd.DataFrame(evaluation\_results)

print(eval\_df)

**9. Feature Importance**

For the best-performing model (in this case, Random Forest), we visualized the **feature importance** to understand the most influential factors in the prediction.

python

Copy code

import matplotlib.pyplot as plt

import numpy as np

def plot\_feature\_importance(model, X\_train):

feature\_importances = model.feature\_importances\_

sorted\_idx = np.argsort(feature\_importances)

plt.figure(figsize=(10, 8))

plt.barh(X\_train.columns[sorted\_idx], feature\_importances[sorted\_idx])

plt.xlabel('Feature Importance')

plt.title('Random Forest Feature Importance')

plt.show()

plot\_feature\_importance(best\_rf\_model, X\_train)

**10. Conclusion**

After evaluating several machine learning models, the **Random Forest** and **Gradient Boosting** models provided the best performance after hyperparameter tuning. The visualizations provided insight into feature importance, and scaling methods ensured that the models had better performance with high-dimensional data.