|  |  |
| --- | --- |
| Name: SRUTHI V. S.  Course: Data Science | Batch: Weekend Batch  Mini Project 3 |
| 📄 DS Project Documentation: Amazon Delivery Time Prediction | |

**1. Project Overview**

This project aims to predict delivery times for e-commerce orders using various features related to orders, delivery agents, and delivery conditions. Predicting accurate delivery times improves customer satisfaction and operational efficiency.

**Step-by-Step Instructions for Delivery Time Prediction Project:**

* Loading and Inspecting Data: Loaded the dataset into a Pandas DataFrame and inspected for nulls, duplicates, and inconsistencies.
* Handling Missing Values: Removed rows with missing critical values such as Order\_Time.
* Duplicates: Dropped duplicate rows to ensure data quality.
* Feature Engineering:
* Calculated distance (in km) between store and drop locations using Haversine formula.
* Extracted time-based features like hour of day and day of week from order timestamps.
* Computed pickup delay as minutes between order time and pickup time.

**Step 1: Installing all necessary libraries:**

!pip install geopy mlflow scikit-learn pandas matplotlib seaborn

**Step 2: Load and Preview Dataset**

from google.colab import files

uploaded = files.upload()

**# Load File:**

import pandas as pd

df = pd.read\_csv('amazon\_delivery.csv')

df.head()

**Data Preprocessing:**

Data preparation began with loading the dataset into a Pandas DataFrame in Google Colab. Initial inspection identified missing values, especially in Order\_Time, which were removed to ensure consistency. Duplicate entries were also eliminated. New features were engineered to enrich the dataset. Specifically, we computed Distance\_km using the Haversine formula based on geographic coordinates, extracted time-based features such as Order\_Hour and Order\_DayOfWeek, and calculated Pickup\_Delay\_Minutes by finding the time difference between order and pickup times.

**Step 3: Data Preprocessing and Feature Engineering**

import numpy as np

from geopy.distance import geodesic

**# Calculate geospatial distance**

def calculate\_distance(row):

store\_coords = (row['Store\_Latitude'], row['Store\_Longitude'])

drop\_coords = (row['Drop\_Latitude'], row['Drop\_Longitude'])

return geodesic(store\_coords, drop\_coords).km

df['Distance\_km'] = df.apply(calculate\_distance, axis=1)

**# Convert time columns**

df['Order\_Date'] = pd.to\_datetime(df['Order\_Date'])

df['Order\_Time'] = pd.to\_datetime(df['Order\_Time'])

df['Pickup\_Time'] = pd.to\_datetime(df['Pickup\_Time'])

**# Time-based features**

df['Order\_Hour'] = df['Order\_Time'].dt.hour

df['Order\_DayOfWeek'] = df['Order\_Date'].dt.dayofweek

df['Pickup\_Delay\_Minutes'] = (df['Pickup\_Time'] - df['Order\_Time']).dt.total\_seconds() / 60

Previous code threw an error due to NaN values in Order\_Time, so we have checked the number of NaN values and decided to remove

**Updated Step: Data Preprocessing & Feature Engineering (with NaN handling)**

import numpy as np

import pandas as pd

from geopy.distance import geodesic

**# Drop rows with any NaN in critical datetime columns**

df = df.dropna(subset=['Order\_Date', 'Order\_Time', 'Pickup\_Time',

'Store\_Latitude', 'Store\_Longitude',

'Drop\_Latitude', 'Drop\_Longitude'])

**# Convert to datetime**

df['Order\_Date'] = pd.to\_datetime(df['Order\_Date'], errors='coerce')

df['Order\_Time'] = pd.to\_datetime(df['Order\_Time'], errors='coerce')

df['Pickup\_Time'] = pd.to\_datetime(df['Pickup\_Time'], errors='coerce')

**# Drop any remaining invalid datetime rows**

df = df.dropna(subset=['Order\_Date', 'Order\_Time', 'Pickup\_Time'])

**# Calculate geospatial distance**

def calculate\_distance(row):

store\_coords = (row['Store\_Latitude'], row['Store\_Longitude'])

drop\_coords = (row['Drop\_Latitude'], row['Drop\_Longitude'])

return geodesic(store\_coords, drop\_coords).km

df['Distance\_km'] = df.apply(calculate\_distance, axis=1)

**# Feature Engineering - Extract time-based features**

df['Order\_Hour'] = df['Order\_Time'].dt.hour

df['Order\_DayOfWeek'] = df['Order\_Date'].dt.dayofweek

df['Pickup\_Delay\_Minutes'] = (df['Pickup\_Time'] - df['Order\_Time']).dt.total\_seconds() / 60

# Preview

df[['Distance\_km', 'Order\_Hour', 'Order\_DayOfWeek', 'Pickup\_Delay\_Minutes']].head()

**What This Fixes:**

* **Drops rows** where essential date/time or location fields are missing.
* Uses errors='coerce' to convert invalid date strings to NaT, then drops them.
* Ensures all datetime math (e.g. pickup delay) works without throwing errors.

**Step 4: Data Cleaning**

**# Start with the original DataFrame**

df\_cleaned = df.copy()

**# Remove duplicates**

df\_cleaned.drop\_duplicates(inplace=True)

**# Handle missing values**

df\_cleaned.dropna(inplace=True) # Apply dropna directly to df\_cleaned

**# The one-hot encoding will be done in a separate cell after visualizations**

**Step 5: Exploratory Data Analysis (EDA)**

Explored the relationships between delivery time and various features. Visualizations showed that traffic congestion, poor weather conditions, and longer distances were associated with longer delivery times. Agent-related features such as age and rating were also examined, though they had a moderate influence. Box plots, bar charts, and correlation heatmaps helped identify key drivers of delivery delay and guided feature selection for model training.

import matplotlib.pyplot as plt

import seaborn as sns

**# Set default Seaborn style**

sns.set(style="whitegrid")

**# 1. Distribution of Delivery Time**

plt.figure(figsize=(10, 6))

sns.histplot(df\_cleaned['Delivery\_Time'], bins=30, kde=True)

plt.title("Distribution of Delivery Times")

plt.xlabel("Delivery Time (hours)")

plt.ylabel("Frequency")

plt.show()

**# 2. Boxplot: Traffic vs Delivery Time**

plt.figure(figsize=(10, 6))

sns.boxplot(x='Traffic', y='Delivery\_Time', data=df\_cleaned)

plt.title("Impact of Traffic on Delivery Time")

plt.show()

**# 3. Boxplot: Weather vs Delivery Time**

plt.figure(figsize=(10, 6))

sns.boxplot(x='Weather', y='Delivery\_Time', data=df\_cleaned)

plt.title("Impact of Weather on Delivery Time")

plt.show()

**# 4. Boxplot: Area vs Delivery Time**

plt.figure(figsize=(10, 6))

sns.boxplot(x='Area', y='Delivery\_Time', data=df\_cleaned)

plt.title("Impact of Area Type on Delivery Time")

plt.show()

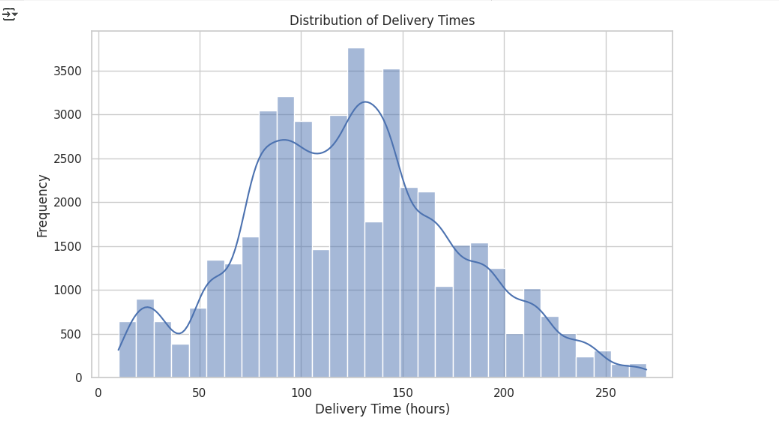
**# 5. Correlation heatmap (numeric features only)**

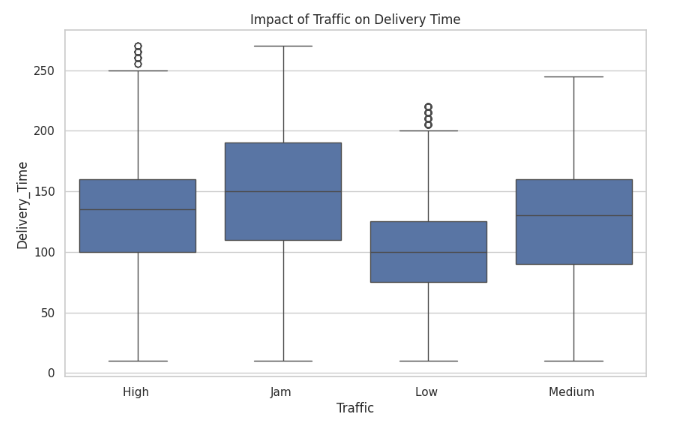
plt.figure(figsize=(12, 8))

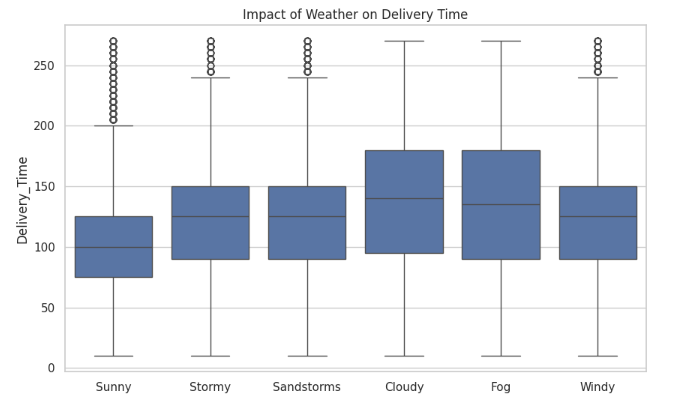
sns.heatmap(df\_cleaned.select\_dtypes(include=['float64', 'int64']).corr(), annot=True, cmap='coolwarm')

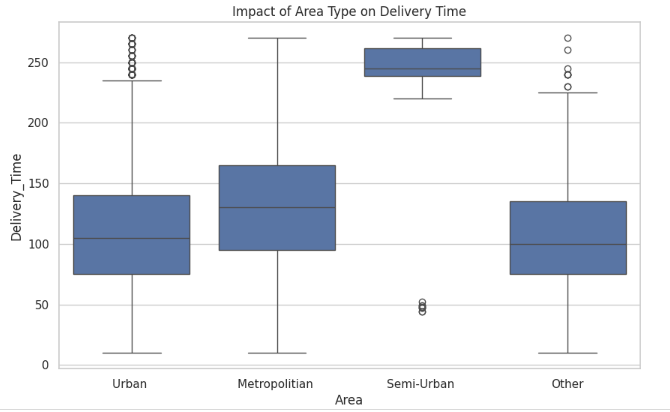
plt.title("Correlation Between Numerical Features")

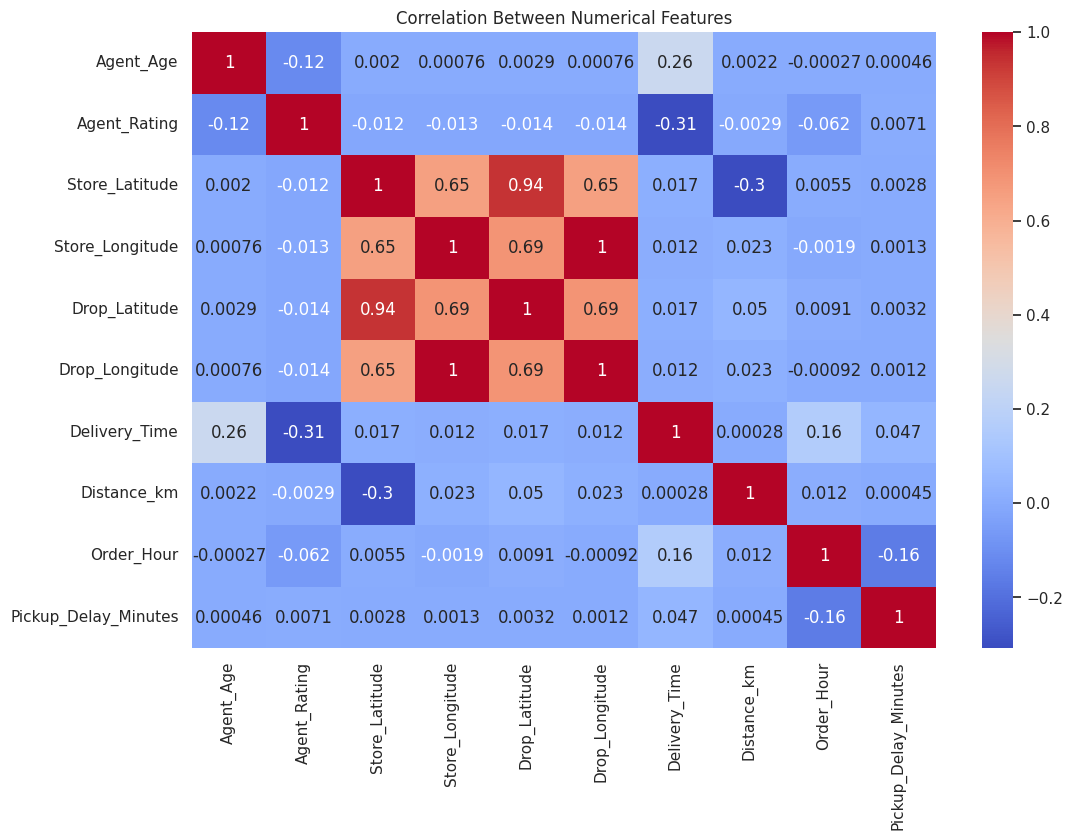
plt.show()

****

****

****

****



**Step 6: One Hot Encoding**

Categorical variables such as Weather, Traffic, Vehicle, Area, and Category were converted to numerical format using one-hot encoding.

**# Remove leading/trailing spaces (important for consistent column names)**

df\_cleaned.columns = df\_cleaned.columns.str.strip()

**# Apply one-hot encoding**

df\_encoded = pd.get\_dummies(df\_cleaned, columns=['Weather', 'Traffic', 'Vehicle', 'Area', 'Category'], drop\_first=True)

**# Now Let us a. Split the data into training and testing sets b.Train multiple regression models and c. Track them using MLflow.**

**Step 7 : Split the Data**

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

**# Define features and target**

X = df\_encoded.drop(['Order\_ID', 'Delivery\_Time', 'Order\_Date', 'Order\_Time', 'Pickup\_Time'], axis=1)

y = df\_encoded['Delivery\_Time']

**# Split into train and test sets (80% train, 20% test)**

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

print("Train shape:", X\_train.shape)

print("Test shape:", X\_test.shape)

**# Import necessary packages**

Import numpy as np

import mlflow

import mlflow.sklearn

from sklearn.linear\_model import LinearRegression

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

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

**Step 8: Train & Track Models with MLflow**

def evaluate\_model(model, name):

    mlflow.set\_experiment("DeliveryTimePrediction")

    with mlflow.start\_run(run\_name=name):

        model.fit(X\_train, y\_train)

        preds = model.predict(X\_test)

**# Metrics**

        mse = mean\_squared\_error(y\_test, preds) # Calculate MSE first

        rmse = np.sqrt(mse) # Then calculate RMSE by taking the square root of MSE

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

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

**# Log to MLflow**

        mlflow.log\_param("model", name)

        mlflow.log\_metric("rmse", rmse)

        mlflow.log\_metric("mae", mae)

        mlflow.log\_metric("r2", r2)

        mlflow.sklearn.log\_model(model, name + "\_model")

        print(f"{name} Results: RMSE={rmse:.2f}, MAE={mae:.2f}, R²={r2:.2f}")

**# Train models**

evaluate\_model(LinearRegression(), "LinearRegression")

evaluate\_model(RandomForestRegressor(n\_estimators=100, random\_state=42), "RandomForest")

evaluate\_model(GradientBoostingRegressor(n\_estimators=100, random\_state=42), "GradientBoosting")

**Result:**

**2025/06/07 15:16:27 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.**

**LinearRegression Results: RMSE=32.17, MAE=25.51, R²=0.61**

**2025/06/07 15:17:26 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.**

**RandomForest Results: RMSE=22.83, MAE=17.51, R²=0.80**

**2025/06/07 15:17:41 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.**

**GradientBoosting Results: RMSE=25.26, MAE=19.59, R²=0.76**

**Step 9: Determine the Best model:**

**Let us now, Analyse the model performance by Comparing the metrics (RMSE, MAE, R²) for the Linear Regression, Random Forest, and Gradient Boosting models to determine which one performed best on your test data.**

import numpy as np # Import numpy

**# Define the models to train and evaluate**

models = {

    "LinearRegression": LinearRegression(),

    "RandomForest": RandomForestRegressor(n\_estimators=100, random\_state=42),

    "GradientBoosting": GradientBoostingRegressor(n\_estimators=100, random\_state=42)

**}**

Results = []

for 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) **# Calculate MSE**

    rmse = np.sqrt(mse) **# Calculate RMSE by taking the square root of MSE**

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

    r2 = r2\_score(y\_test, y\_pred)

    Results.append({

        "Model": name,

        "RMSE": round(rmse, 2),

        "MAE": round(mae, 2),

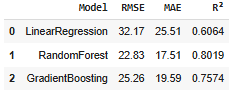
        "R²": round(r2, 4)

    })

**# Convert to DataFrame for comparison**

results\_df = pd.DataFrame(Results)

results\_df

****

**Model Performance Analysis:**

* **RMSE (Root Mean Squared Error): Penalizes large errors more heavily — lower is better.**
* **MAE (Mean Absolute Error): Straight average of absolute errors — lower is better.**
* **R² (R-squared): How well the model explains variance in the target — higher is better.**

**Analysis**

**Linear Regression:**

* **Worst performance on all three metrics.**
* **High errors and poor explanatory power.**
* **Not suitable for this use case.**

**Gradient Boosting:**

* **Performs significantly better than Linear Regression.**
* **But slightly worse than Random Forest in all metrics.**

**Random Forest:**

* **Best model overall:**
  + **Lowest RMSE (22.83): Most accurate on average.**
  + **Lowest MAE (17.51): Least average error.**
  + **Highest R² (0.8019): Explains ~80% of the variability in delivery times.**

**Best Model: Random Forest Regressor**

**Recommendation: Use the Random Forest Regressor for your prediction system.**

**Three regression models were developed to predict delivery time: Linear Regression, Random Forest Regressor, and Gradient Boosting Regressor. The data was split into training and testing sets. Each model was trained on the processed features and evaluated using metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R² (coefficient of determination). The Random Forest Regressor emerged as the best performer, offering the highest accuracy and lowest error.**

**Step 10: Let us save the trained Random Forest model to a .pkl file so you can load it later in your Streamlit app**

**import joblib**

**# Assuming 'rf\_model' is your trained Random Forest model**

rf\_model = models['RandomForest'] **# or wherever you have your fitted model**

**# Save model to file**

joblib\_file = "random\_forest\_model.pkl"

joblib.dump(rf\_model, joblib\_file)

print(f"Model saved to {joblib\_file}")

**Install streamlit:**

!pip install streamlit

**Step 11: Save the file in local environment:**

streamlit\_code = """

import streamlit as st

import joblib

import numpy as np

# Load your saved model

model = joblib.load('random\_forest\_model.pkl')

st.title("E-commerce Delivery Time Prediction")

# Numeric inputs

agent\_age = st.number\_input("Agent Age", min\_value=18, max\_value=80, step=1)

agent\_rating = st.number\_input("Agent Rating (1-5)", min\_value=1.0, max\_value=5.0, step=0.1)

store\_lat = st.number\_input("Store Latitude", format="%.6f")

store\_lon = st.number\_input("Store Longitude", format="%.6f")

drop\_lat = st.number\_input("Drop Latitude", format="%.6f")

drop\_lon = st.number\_input("Drop Longitude", format="%.6f")

distance\_km = st.number\_input("Distance (km)", min\_value=0.0, step=0.1)

order\_hour = st.number\_input("Order Hour (0-23)", min\_value=0, max\_value=23, step=1)

order\_day = st.number\_input("Order Day of Week (0=Monday)", min\_value=0, max\_value=6, step=1)

pickup\_delay = st.number\_input("Pickup Delay (minutes)", min\_value=0, step=1)

# Weather options

weather\_options = ['Fog', 'Sandstorms', 'Stormy', 'Sunny', 'Windy']

weather = st.selectbox("Weather Condition", weather\_options)

weather\_fog = 1 if weather == 'Fog' else 0

weather\_sandstorms = 1 if weather == 'Sandstorms' else 0

weather\_stormy = 1 if weather == 'Stormy' else 0

weather\_sunny = 1 if weather == 'Sunny' else 0

weather\_windy = 1 if weather == 'Windy' else 0

# Traffic options

traffic\_options = ['Jam', 'Low', 'Medium']

traffic = st.selectbox("Traffic Condition", traffic\_options)

traffic\_jam = 1 if traffic == 'Jam' else 0

traffic\_low = 1 if traffic == 'Low' else 0

traffic\_medium = 1 if traffic == 'Medium' else 0

# Vehicle options

vehicle\_options = ['scooter', 'van']

vehicle = st.selectbox("Vehicle Type", vehicle\_options)

vehicle\_scooter = 1 if vehicle == 'scooter' else 0

vehicle\_van = 1 if vehicle == 'van' else 0

# Area options

area\_options = ['Other', 'Semi-Urban', 'Urban']

area = st.selectbox("Delivery Area", area\_options)

area\_other = 1 if area == 'Other' else 0

area\_semiurban = 1 if area == 'Semi-Urban' else 0

area\_urban = 1 if area == 'Urban' else 0

# Category options

category\_options = ['Books', 'Clothing', 'Cosmetics', 'Electronics', 'Grocery', 'Home', 'Jewelry', 'Kitchen', 'Outdoors',

'Pet Supplies', 'Shoes', 'Skincare', 'Snacks', 'Sports', 'Toys']

category = st.selectbox("Product Category", category\_options)

category\_books = 1 if category == 'Books' else 0

category\_clothing = 1 if category == 'Clothing' else 0

category\_cosmetics = 1 if category == 'Cosmetics' else 0

category\_electronics = 1 if category == 'Electronics' else 0

category\_grocery = 1 if category == 'Grocery' else 0

category\_home = 1 if category == 'Home' else 0

category\_jewelry = 1 if category == 'Jewelry' else 0

category\_kitchen = 1 if category == 'Kitchen' else 0

category\_outdoors = 1 if category == 'Outdoors' else 0

category\_pet\_supplies = 1 if category == 'Pet Supplies' else 0

category\_shoes = 1 if category == 'Shoes' else 0

category\_skincare = 1 if category == 'Skincare' else 0

category\_snacks = 1 if category == 'Snacks' else 0

category\_sports = 1 if category == 'Sports' else 0

category\_toys = 1 if category == 'Toys' else 0

features = np.array([[

agent\_age,

agent\_rating,

store\_lat,

store\_lon,

drop\_lat,

drop\_lon,

distance\_km,

order\_hour,

order\_day,

pickup\_delay,

weather\_fog,

weather\_sandstorms,

weather\_stormy,

weather\_sunny,

weather\_windy,

traffic\_jam,

traffic\_low,

traffic\_medium,

vehicle\_scooter,

vehicle\_van,

area\_other,

area\_semiurban,

area\_urban,

category\_books,

category\_clothing,

category\_cosmetics,

category\_electronics,

category\_grocery,

category\_home,

category\_jewelry,

category\_kitchen,

category\_outdoors,

category\_pet\_supplies,

category\_shoes,

category\_skincare,

category\_snacks,

category\_sports,

category\_toys

]])

if st.button("Predict Delivery Time"):

prediction = model.predict(features)

st.success(f"Predicted Delivery Time: {prediction[0]:.2f} hours")

"""

with open("app.py", "w") as file:

file.write(streamlit\_code)

print("app.py has been saved!")

**Next steps:**

* **Saved app.py locally** by copying the code from above into a text editor and save it as app.py.
* **Placed saved model file** (e.g., random\_forest\_model.pkl) in the same folder as app.py.
* Now its time to **Run the Streamlit app**:

**Run:**

pip install streamlit joblib numpy

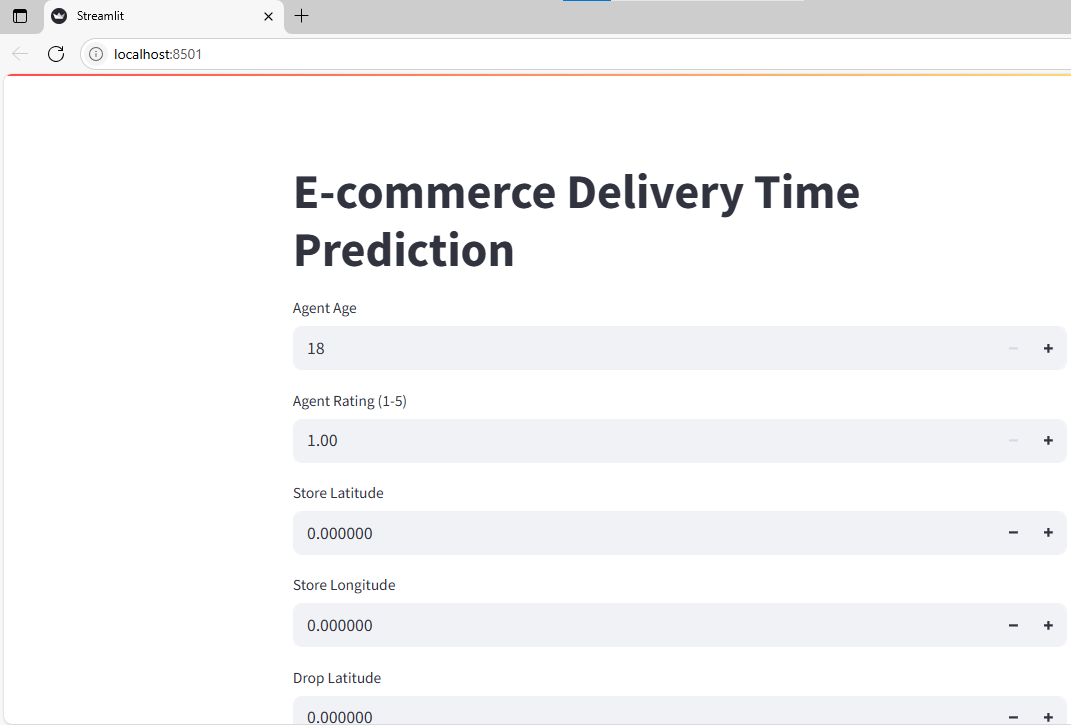
pip install scikit-learn

Cd C:\Users\LENOVO\Documents\Data Science Project\Mini Project 3 - Amazon Delivery Time Prediction

**Run:**

streamlit run app.py

Your browser will open automatically showing your app at <http://localhost:8501>

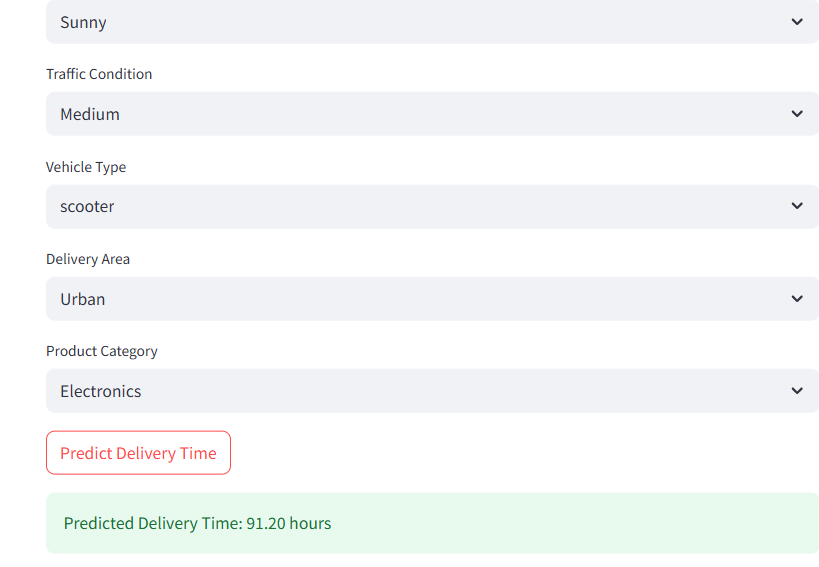


Selected the below values on the app to Predict delivery time:

| **Field** | **Sample Value** |
| --- | --- |
| **Agent Age** | 30 |
| **Agent Rating** | 4.5 |
| **Store Latitude** | 40.712776 |
| **Store Longitude** | -74.005974 |
| **Drop Latitude** | 40.730610 |
| **Drop Longitude** | -73.935242 |
| **Distance (km)** | 8.5 |
| **Order Hour** | 14 |
| **Order Day of Week** | 3 |
| **Pickup Delay (minutes)** | 10 |

**Dropdown selections:**

* **Weather:** Sunny
* **Traffic:** Medium
* **Vehicle:** Van
* **Area:** Urban
* **Category:** Electronics



A user-friendly prediction interface was built using **Streamlit**. This web app allows users to input relevant order and delivery information through form fields and dropdowns. Upon clicking the **Predict Delivery Time** button, the app loads the saved model and displays the predicted delivery time. The interface supports numerical inputs for features like agent age, distance, and pickup delay, and dropdowns for weather, traffic, vehicle type, and category.