Full Documentation of Data_Science_1.ipynb

Introduction

This notebook represents a structured approach to data science, focusing on exploratory data analysis (EDA), data preprocessing, and baseline model building. The dataset used spans from 2008 to 2019, loaded from a CSV file located in Google Drive. The goal of this analysis is to understand the data, clean and prepare it, and finally apply a simple machine learning model (Linear Regression) to gain predictive insights.

Importing Libraries

```
import numpy as np
```

This line imports NumPy, a core library for numerical operations in Python. It provides support for arrays, matrices, and numerous mathematical functions.

```
from google.colab import drive
drive.mount('/content/drive')
```

These lines are specific to Google Colab. The code mounts the user's Google Drive, allowing access to files stored there, such as datasets.

```
import pandas as pd

df = pd.read_csv('/content/drive/MyDrive/Combined Data_2008-2019 - Sheet1.csv')
```

Pandas is imported for data manipulation, and the dataset is loaded from the mounted Google Drive. It is stored in a DataFrame called `df`.

```
print(f"Initial shape: {df.shape}")
display(df.head())
```

This code checks the size of the dataset and displays the first five rows. It helps confirm successful loading and offers a quick glance at the structure.

Dataset Overview

```
# Step 1: Show basic info
print("\n--- Data Info ---")
df.info()
```

The `df.info()` function gives an overview of each column's data type, number of non-null values, and memory usage. This is essential for detecting missing data and verifying the format of each column.

Summary Statistics

```
# Step 2: Describe numeric columns
print("\n--- Summary Statistics ---")
df.describe()
```

This command provides summary statistics for all numerical columns, including count, mean, standard deviation, minimum, and quartiles. These insights help understand the distribution and variance of the data.

Detecting Missing and Duplicate Values

```
# Step 3: Check for missing values
print("\n--- Missing Values ---")
print(df.isnull().sum())
```

This block calculates how many missing (`NaN`) values are present in each column. This is a necessary step before deciding how to impute or handle missing data.

```
# Step 4: Check for duplicates
print("\n--- Duplicate Rows ---")
print(df.duplicated().sum())
```

This code identifies and counts any rows that are duplicated, which could distort analysis and model training.

Column Names and Data Types

```
# Step 5: Print column names
print("\n--- Column Names ---")
print(df.columns)
```

This simply prints all the column names, which can be helpful for renaming, referencing, or selecting columns for further analysis.

```
# Step 6: Check data types
print("\n--- Data Types ---")
print(df.dtypes)
```

Data types determine how operations can be performed on columns. For example, numerical operations cannot be performed on columns with object (string) data types without conversion.

Correlation Analysis

```
# Step 7: Correlation matrix
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap")
plt.show()
```

This block calculates and visualizes the correlation matrix among numerical variables using a heatmap. It is useful for detecting strong positive or negative correlations, which can inform feature selection.

Distribution of a Numerical Column

```
# Step 8: Distribution plot
sns.histplot(df['some_numerical_column'], kde=True)
plt.title("Distribution of a Numerical Column")
plt.show()
```

This code visualizes the distribution of a numerical variable. The KDE (Kernel Density Estimation) curve overlays a smoothed line to represent the probability density function.

Count Plot for a Categorical Variable

```
# Step 9: Count plot
sns.countplot(x='some_category_column', data=df)
plt.title("Count of Categorical Values")
plt.xticks(rotation=45)
plt.show()
```

Count plots are used to visualize the number of observations in each category, helping detect imbalance in classification tasks.

Box Plot for Outlier Detection

```
# Step 10: Boxplot
sns.boxplot(x='some_category_column', y='some_numerical_column', data=df)
plt.title("Boxplot Grouped by Category")
plt.xticks(rotation=45)
plt.show()
```

Boxplots help visualize the spread and identify outliers by grouping numerical data by categorical labels.

Scatter Plot for Relationships

```
# Step 11: Scatter plot
sns.scatterplot(x='feature1', y='feature2', data=df)
plt.title("Scatter Plot Between Two Features")
plt.show()
```

Scatter plots show how two continuous features relate to each other. They help identify clusters, linearity, or non-linear trends.

Grouped Aggregation

```
# Step 12: Grouped mean
print(df.groupby('some_category_column').mean())
```

This code groups the dataset by a categorical variable and computes the mean for each group, providing insights into how different groups affect the numerical outcome.

Handling Missing Values

```
# Step 13: Fill missing values
df.fillna(method='ffill', inplace=True)
```

Forward fill propagates the last valid value forward. This is useful in sequential datasets, like time series.

Label Encoding

```
# Step 14: Label Encoding
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['encoded_column'] = le.fit_transform(df['some_category_column'])
```

This converts categorical variables into numeric labels. Each unique category gets a number. It's useful for models that cannot handle strings.

Feature Scaling

```
# Step 15: Feature scaling
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_data = scaler.fit_transform(df[['feature1', 'feature2']])
```

StandardScaler normalizes features so they have zero mean and unit variance. This is important for algorithms that use distance metrics or gradient-based optimization.

Train-Test Split

```
# Step 16: Split dataset
from sklearn.model_selection import train_test_split

X = df.drop('target_column', axis=1)
y = df['target_column']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

The data is split into training and testing subsets. The model is trained on 80% of the data and evaluated on the remaining 20% to ensure generalization.

Linear Regression Model

```
# Step 17: Linear Regression
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

This section trains a linear regression model. The algorithm finds the best-fit line that minimizes the sum of squared errors between the actual and predicted values.

Model Evaluation

```
# Step 18: Evaluation metrics
from sklearn.metrics import mean_squared_error, r2_score

mse = mean_squared_error(y_test, predictions)
r2 = r2_score(y_test, predictions)

print("MSE:", mse)
print("R2 Score:", r2)
```

Mean Squared Error (MSE) and R-squared (R^2) are used to evaluate the model's performance. Lower MSE and higher R^2 values indicate better performance.

Actual vs Predicted Visualization

```
# Step 19: Plot predictions
plt.scatter(y_test, predictions)
plt.xlabel('Actual')
plt.ylabel('Predicted')
```

```
plt.title("Actual vs Predicted")
plt.show()
```

This scatter plot helps visually assess how closely the predicted values match the actual values. A perfect model would have all points lying on the diagonal line.

Conclusion

The notebook successfully walks through the stages of a typical data science project. Starting from data loading, the notebook performs basic diagnostics, handles missing and duplicate values, explores relationships through visualizations, encodes and scales features, splits the data, builds a regression model, and evaluates its performance. Each step follows standard best practices and prepares the ground for more advanced modeling and tuning in future iterations.