Customer Churn Prediction: Detailed Explanation

1. Importing Libraries

The project begins by importing essential libraries for data manipulation, visualization, and machine learning.

- `numpy` and `pandas`: Handle numerical operations and tabular data.
- `matplotlib.pyplot` and `seaborn`: Create visualizations for EDA.
- `sklearn` modules: Perform preprocessing, model training, and evaluation.
- `imblearn.over_sampling.SMOTE`: Balance imbalanced datasets through up-sampling.
- `pickle`: Save and load models and preprocessing encoders.

Code Block:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import pickle
```

These libraries form the backbone of the project, providing tools for data processing, visualization, and machine learning tasks.

Example Usage:
- `pandas`: Read and preprocess the dataset.
- `seaborn`: Create visualizations like histograms and heatmaps to identify data trends.

- `LabelEncoder`: Encode categorical variables into numerical format for model compatibility.
- `SMOTE`: Generate synthetic samples to address class imbalance.

2. Data Loading and Exploration

Loading the dataset is the first step in any machine learning workflow. It involves inspecting the dataset's dimensions,
structure, and identifying potential issues like missing values or incorrect data types.

Code Block:

```
df = pd.read_csv("data.csv")
df.shape  # Returns the number of rows and columns
df.head()  # Displays the first 5 rows of the dataset
pd.set_option("display.max_columns", None)  # Ensures all columns are visible in console
outputs
df.info()  # Displays column names, data types, and non-null counts
```

- **`pd.read_csv("data.csv")`**: Loads the dataset into a pandas DataFrame for analysis.
- **`df.shape`**: Displays the number of rows (data points) and columns (features) in the dataset.
- **`df.info()`**: Shows the structure of the dataset, including data types and the presence of null values.

Why use `pd.set_option("display.max_columns", None)`?
- This ensures all columns are displayed, which is essential for wide datasets with many features.

3. Dropping Irrelevant Features

Columns that do not contribute meaningful information to the prediction task should be removed. For instance, `customerID`
is a unique identifier and does not contain any predictive value.

Code Block:

```
df = df.drop(columns=['customerID'])
```

- **Why drop `customerID`?**
  - It is a unique identifier that does not influence churn prediction.
  - Removing such columns reduces noise in the dataset and simplifies preprocessing.

4. Visualizing Data: Histograms and Box Plots

Histograms:

- Used to visualize the frequency distribution of numerical features.
- Help identify central tendency (mean, median), spread, and skewness.

Code Block:

```
def plot_hist(df, column_name):
    plt.figure(figsize=(5,3))
    sns.histplot(df[column_name], kde=True)
    plt.title(f"{column_name} distribution")
    col_mean = df[column_name].mean()
    col_median = df[column_name].median()
    plt.axvline(col_mean, color='red', linestyle='--', label='Mean')
    plt.axvline(col_median, color='green', linestyle='-', label='Median')
    plt.legend()
    plt.show()

for col in ['tenure', 'MonthlyCharges', 'TotalCharges']:
    plot_hist(df, col)
```

- **`sns.histplot()`**: Creates a histogram with an optional KDE curve to visualize the data distribution.
- **`axvline()`**: Plots vertical lines to indicate the mean and median.

Box Plots:

- Summarize numerical data using quartiles.
- Highlight outliers, which can affect model performance.

Code Block:

```
def plot_boxplot(df, column_name):
    plt.figure(figsize=(5, 3))
    sns.boxplot(y=df[column_name])
    plt.title(f'Box Plot of {column_name}')
    plt.ylabel(column_name)
    plt.show()

for col in ['tenure', 'MonthlyCharges', 'TotalCharges']:
    plot_boxplot(df, col)
```

- **`sns.boxplot()`**: Visualizes data spread and outliers.
- **Why use these plots?**
  - Histograms help detect skewness and transformations required.
  - Box plots highlight potential outliers, which might need special handling.

5. Sampling Techniques in Machine Learning

Sampling is a crucial step in machine learning, particularly when dealing with imbalanced datasets. In such datasets, one class
(e.g., "Churned") is underrepresented compared to the other class (e.g., "Not Churned"). This imbalance can lead to biased
models that favor the majority class.

There are two primary sampling techniques:

1. **Up-Sampling (Over-Sampling)**:
   - Increases the number of samples in the minority class.
   - Methods include duplicating existing samples or generating synthetic samples.
   - The most common approach is SMOTE (Synthetic Minority Oversampling Technique).

2. **Down-Sampling (Under-Sampling)**:

   - Reduces the number of samples in the majority class by randomly discarding some data points.

   - This ensures both classes have equal representation in the dataset.

Why Use Sampling?

Sampling helps address class imbalance, which can skew model predictions. Without balancing, models may fail to learn
patterns in the minority class, leading to poor performance on that class.

Comparison of Up-Sampling and Down-Sampling:

- **Up-Sampling**:
  - Preserves all the data from the majority class.
  - Can lead to overfitting if synthetic samples are too similar to existing ones.
  - Suitable for small datasets.

- **Down-Sampling**:
  - Reduces training data size, improving computational efficiency.
  - Risks losing valuable information by discarding majority class samples.
  - Suitable for large datasets.

Which is Better?

- **Up-Sampling** (e.g., SMOTE) is generally preferred when working with small or moderately sized datasets, as it avoids data loss.
- **Down-Sampling** is useful for very large datasets where removing some data points does not significantly impact model performance.

Code Example: Up-Sampling Using SMOTE:

```
from imblearn.over_sampling import SMOTE

# Apply SMOTE to training data
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
```

```
# Display class distribution after up-sampling
print("Class distribution after SMOTE:")
print(y_train_smote.value_counts())
```

Explanation:

- **`SMOTE`**: Generates synthetic samples for the minority class by interpolating between existing samples.
- **`fit_resample()`**: Applies SMOTE to the training data, creating a balanced dataset.
- **Why SMOTE?**
  - Unlike random oversampling, SMOTE reduces the risk of overfitting by generating synthetic, rather than duplicate, samples.

By addressing class imbalance, sampling techniques improve the model's ability to generalize across all classes.