Telco-Customer-Churn

Group members:

1: Ahmed Shaheer

2: Daniyal Jawad

3: Sarmad Siddique

Documentation

Step 1: Data Exploration and Preprocessing

This section details the process of exploring and preparing the dataset for further analysis. We'll be using the pandas library for data manipulation.

1.1. Loading the Dataset:

```
import pandas as pd
f = pd.read_csv("WA_Fn-UseC_-Telco-Customer-Churn.csv")
```

This code snippet imports the `pandas` library and reads the CSV file named "WA_Fn-UseC_-Telco-Customer-Churn.csv" into a DataFrame object named df. A DataFrame is a tabular data structure similar to a spreadsheet.

1.2. Initial Exploration:

```
# Display the first few rows of the data
print(df)

# Get information about the DataFrame, including data types and missing values
df.info()

# Get summary statistics for numerical columns
df.describe()
```

These lines perform some initial exploration of the data:

1: `df` : This simply prints the first few rows of the DataFrame to get a glimpse of the data's content.

2: `df.info()` : This provides information about the DataFrame, such as the number of rows and columns, data types of each column, and the presence of any missing values.

3: `df.describe()` : This calculates summary statistics for numerical columns in the DataFrame, including mean, standard deviation, minimum, maximum, and percentiles. This helps understand the distribution of numerical values.

1.3. Data Cleaning and Preprocessing:

```
# Create a copy of the DataFrame to avoid modifying the original data
telco_data = df.copy()

# Convert the 'TotalCharges' column to numeric format (assuming it represents numerical
data)
telco_data.TotalCharges = pd.to_numeric(telco_data.TotalCharges, errors='coerce')

# Check for missing values after conversion (in case conversion failed for some entries)
print(telco_data.isnull().sum())

# Explore rows with missing values in 'TotalCharges' (optional)
telco_data.loc[telco_data ['TotalCharges'].isnull() == True]
```

This part focuses on cleaning and preprocessing the data:

1: `telco_data = df.copy()` : We create a copy of the original DataFrame to avoid making permanent modifications.

2: `telco_data.TotalCharges = pd.to_numeric(...)` : This line attempts to convert the 'TotalCharges' column to a numerical format (e.g., float) if it wasn't already. The `errors='coerce'` argument instructs the function to replace conversion errors with NaN (Not a Number) values.

3: `telco_data.isnull().sum()` : This line checks for missing values (NaN) across all columns in the DataFrame and displays the count of missing values for each column. This helps identify potential issues with missing data.

4: `telco_data.loc[telco_data ['TotalCharges'].isnull() == True]` : This line explores the rows where the 'TotalCharges' value is missing. It utilizes the `.loc` accessor of the DataFrame to filter rows based on a condition. In this case, the condition `telco_data['TotalCharges'].isnull() == True` selects rows where the 'TotalCharges' value is null (missing).

1.4. Removing Missing values

```
telco_data.dropna(how='any', inplace=True)
```

The `dropna` function from the pandas library is used to remove rows containing missing values.

By running this code, you'll effectively remove rows that have missing values in any column of the `telco_data` DataFrame. It's essential to consider the potential impact of removing data on the representativeness and accuracy of your analysis, especially if the number of missing values is significant.

2. Customer Segmentation

This section describes the process of segmenting customers into distinct groups based on their characteristics using K-Means clustering.

```
# Check data shape
telco_data.shape

# Drop irrelevant column (assuming 'customerID' is not required for clustering)
telco_data.drop(['customerID'],axis=1,inplace=True)

# View the first few rows after dropping the column
telco_data.head()

# Check for missing values again
print(telco_data.isnull().sum())
```

These lines perform some initial checks and data cleaning:

1: We check the data shape (`telco_data.shape`) to see the number of rows and columns.

2: The customerID column might not be relevant for clustering customer behavior, so we drop it using `telco_data.drop(...)` .

3: We examine the first few rows (`telco_data.head()`) after dropping the column to verify the changes.

4: Finally, we check for missing values again using print(`telco_data.isnull().sum()`) to ensure there are no missing values that could affect the clustering process.

2.2. Feature Encoding:

```
# Label encoding for categorical features
from sklearn.preprocessing import LabelEncoder

le=LabelEncoder()
telco_data['gender']=le.fit_transform(telco_data['gender'])

# Check data types after encoding
telco_data.dtypes
```

This part focuses on encoding categorical features into numerical representations suitable for K-Means clustering, which typically works better with numerical data.

1:We import the `LabelEncoder` class from scikit-learn.

2:We create a `LabelEncoder` object (`le`) and use it to fit and transform the `gender` column. This assigns numerical labels (0, 1, etc.) to the different categories (e.g., "Male", "Female").

3: We check the data types (`telco_data.dtypes`) again to verify that the `gender` column is now encoded as an integer type.

2.3. Identifying Categorical Features for Encoding:

```
# Identify categorical columns
cols=telco_data.columns
cols
cat_cols=telco_data.select_dtypes(exclude=['int','float']).columns
cat_cols
```

These lines identify the categorical columns that need to be encoded.

1: We get all column names (`telco_data.columns`) and store them in `cols` .

2: We use `telco_data.select_dtypes(exclude=['int','float'])` to filter and select only the columns with data types other than integer or float (which are likely categorical). This gives us the categorical column names in `cat_cols` .

2.4. Encoding Remaining Categorical Features:

```
# Encode remaining categorical features (excluding target variable)
enc_data=list(cat_cols)
enc_data=enc_data[:-1]  # Exclude the last column (assuming it's the target variable)
telco_data[enc_data]=telco_data[enc_data].apply(lambda col:le.fit_transform(col))
telco_data[enc_data].head()

# Check data types again
telco_data.dtypes
```

This part continues the encoding process for the remaining categorical features:

1: We create a list `enc_data` containing the names of all categorical columns except the last one (assuming the last column is the target variable, which we might not want to encode).

2: We apply the `le.fit_transform` function to each column in `enc_data` using `.apply` . This encodes the values in those columns using the previously fitted `LabelEncoder` (`le`).

3: We check the data types again (`telco_data.dtypes`) to confirm that all categorical features are now numerical.

2.5. Handling Missing Values (if applicable):

```
try:
  telco_data['TotalCharges']=pd.to_numeric(telco_data['TotalCharges'])
  # ... handle other missing value cases if necessary
except Exception as e:
  print(e)
```

This code snippet demonstrates how to handle missing values in the `TotalCharges` column (if there were any). It attempts to convert the column to numeric format using `pd.to_numeric` . The `try-except` block is a common way to handle potential errors during data manipulation.

2.6. Encoding Payment Method:

This line encodes the `PaymentMethod` column using the same `LabelEncoder` (`le`). This ensures all features are numerical before applying K-Means clustering.

```
telco_data['PaymentMethod']=le.fit_transform(telco_data['PaymentMethod'])
telco_data.head()
```

2.7. Determining the Optimal Number of Clusters (K):

This section focuses on finding the optimal number of clusters (K) for K-Means clustering using the Elbow Method and Silhouette Score.

2.7.1. The Elbow Method:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score


from scipy.spatial.distance import cdist
from sklearn.cluster import KMeans
distortions = []
mapping1 = {}
k=range(1,15)
for i in k:
  kmeanModel = KMeans(n_clusters=i)
  kmeanModel.fit(telco_data)
  distortions.append(sum(np.min(cdist(telco_data,
kmeanModel.cluster_centers_,'euclidean'),axis=1)) / telco_data.shape[0])
  mapping1[i] = sum(np.min(cdist(telco_data,
kmeanModel.cluster_centers_,'euclidean'),axis=1)) / telco_data.shape[0]
for key,val in mapping1.items():
  print(str(key)+' : '+str(val))
plt.plot(k, distortions, 'bx-')
plt.xlabel('Values of K')
plt.ylabel('Distortion')
plt.title('The Elbow Method using Distortion')
plt.show()
```

The Elbow Method is a visual technique to help identify the optimal number of clusters. Here's a breakdown of the code:

1: We import necessary libraries for data manipulation, plotting, and K-Means clustering.

2: `distortions list`: This list will store the distortion values for different K values.

3: `mapping1` dictionary: This dictionary will temporarily store the distortion values for each K.

4: `k=range(1,15)`: This defines a range of K values (number of clusters) to explore (from 1 to 14).

5: The loop iterates through each K value: A KMeans model (`kmeanModel`) is created with the current K value.

The model is fit to the data (`kmeanModel.fit(telco_data)`).

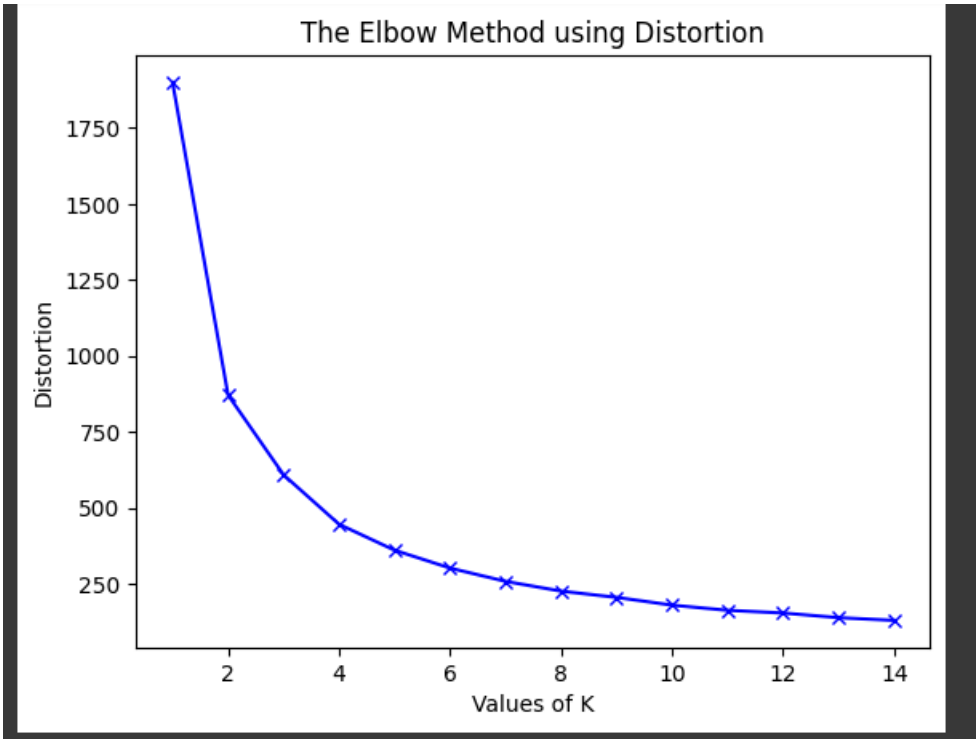
The distortion is calculated using the distance between data points and their closest cluster centers. The `cdist` function calculates pairwise distances between data points and cluster centers. The `min` function finds the minimum distance for each data point to its closest cluster center across all clusters. The sum of these minimum distances is then divided by the total number of data points to get the average distortion for the current K value.

6: `for key, val in mapping1.items()`: This loop iterates over the mapping1 dictionary to print the K value and its corresponding distortion for easier visualization.

7: `plt.plot(k, distortions, 'bx-')`: This line plots the K values on the x-axis and the distortion values on the y-axis. The 'bx-' markers indicate blue x-shaped markers with connecting lines.

8: The plot's title, labels, and legend are set for clarity.

9: `plt.show()`: This displays the Elbow Method plot, which ideally should show an "elbow" where the distortion starts to flatten out as the number of clusters increases. The K value at the "elbow" is often considered a good indicator of the optimal number of clusters.



2.7.2. Silhouette Score:

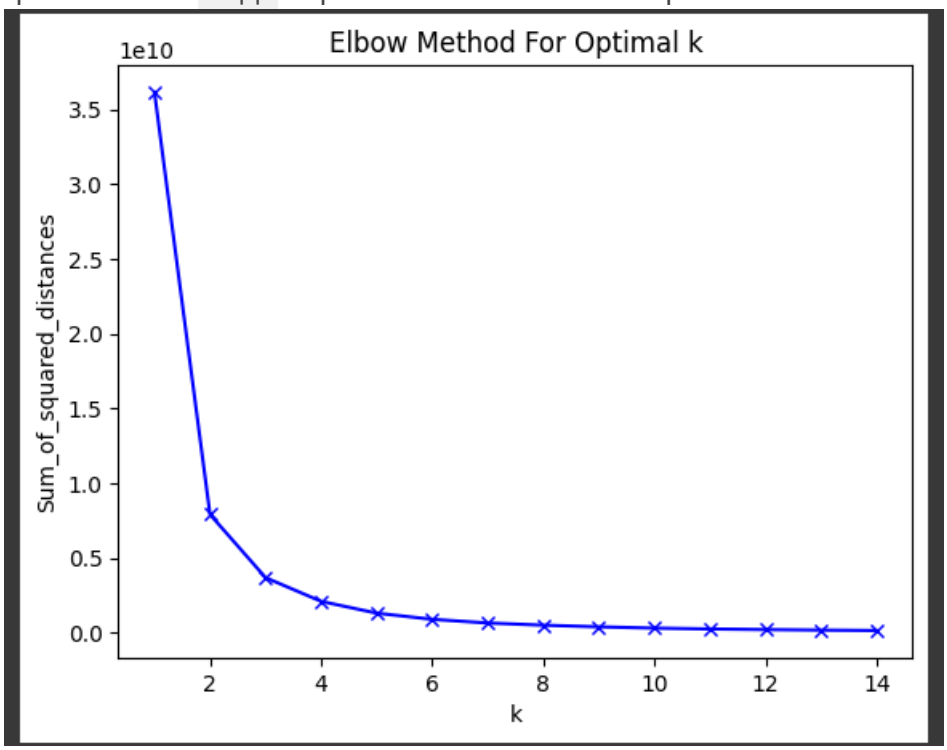
```
Sum_of_squared_distances = []
mapp={}
K = range(1,15)
for k in K:
  km = KMeans(n_clusters=k)
  km = km.fit(telco_data)
  Sum_of_squared_distances.append(km.inertia_)
  mapp[k]=km.inertia_
for key,val in mapp.items():
  print(str(key)+' : '+str(val))
    plt.plot(K, Sum_of_squared_distances, 'bx-')
plt.xlabel('k')
plt.ylabel('Sum_of_squared_distances')
plt.title('Elbow Method For Optimal k')
plt.show()
```

The Silhouette Score is another metric used to assess the quality of clustering. Here's what the code does:

1: `Sum_of_squared_distances` list: This list will store the inertia values for different K values. Inertia is the sum of squared distances between data points and their assigned cluster centers, used as a measure of how well data points are grouped within clusters.

2: `mapp` dictionary: This dictionary stores inertia values for each K.

3: The loop iterates through each K value: A KMeans model (`km`) is created with the current K value. The model is fit to the data (`km.fit(telco_data)`). The inertia (`km.inertia_`) is calculated, representing the sum of squared distances within clusters. The inertia value and K value are stored in the `mapp` dictionary. Similar to the Elbow Method, the loop iterates over `mapp` to print K and inertia values. A plot is created to visualize the inertia for different K values.



2.7.3. Silhouette Score Evaluation:

```
from sklearn.metrics import silhouette_score
try:
  for n_clusters in K:
    clusterer = KMeans (n_clusters=n_clusters).fit(telco_data)
    preds = clusterer.predict(telco_data)
    centers = clusterer.cluster_centers_

    score = silhouette_score (telco_data, preds, metric='euclidean')
    print ("For n_clusters = {}, silhouette score is {})".format(n_clusters, score))
except Exception as e:
  print(e)
```

This part attempts to calculate the Silhouette Score for each K value but might encounter an error (commented out for now). Silhouette Score measures how well data points are assigned to their clusters.

The error might occur if the data is not scaled before applying KMeans clustering. KMeans is sensitive to the scale of features, and features with larger scales can dominate the distance calculations. Scaling the data (e.g., using StandardScaler) can help mitigate this issue.

2.7.4. Evaluating Silhouette Scores:

```
score_list=[]
for n_clusters in range(2,15):
  clusterer = KMeans (n_clusters=n_clusters).fit(telco_data)
  preds = clusterer.predict(telco_data)
  centers = clusterer.cluster_centers_

  score = silhouette_score (telco_data, preds, metric='euclidean')
  score_list.append(score)
  print ("For n_clusters = {}, silhouette score is {})".format(n_clusters, score))

plt.bar(range(2,15),score_list)
plt.show()
```

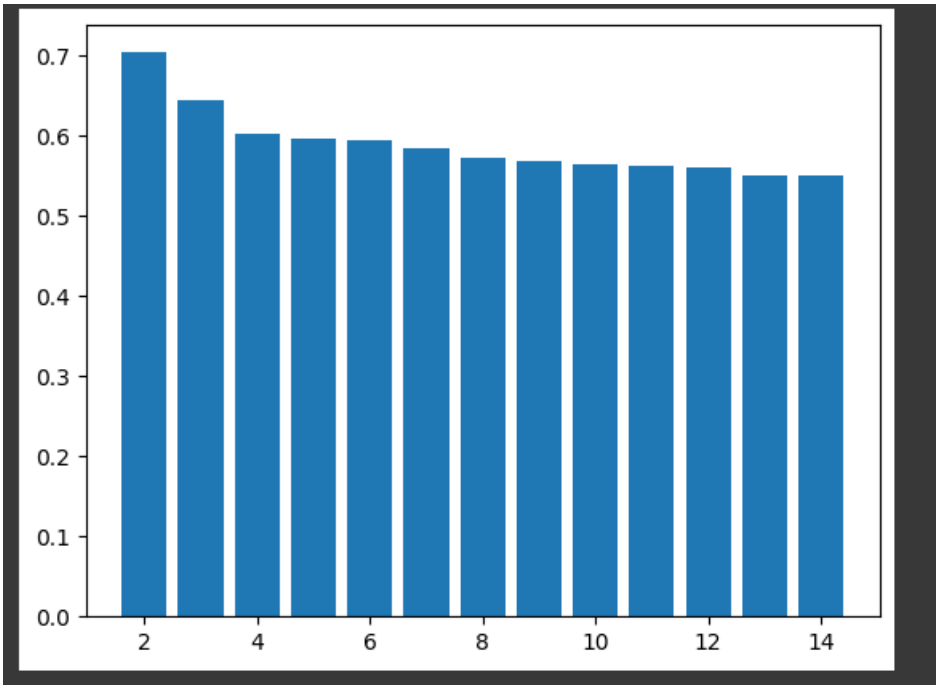
!: The loop iterates through K values from 2 to 14 (inclusive).

2: For each K, similar to the previous attempt, a KMeans model is created, fit, used for prediction, and cluster centers are retrieved.

3: The Silhouette Score is calculated and appended to the `score_list`.

4: The K value and its corresponding score are printed.

5: A bar chart is created to visualize the Silhouette Scores for different K values.



By analyzing the Elbow Method plot, Silhouette Score plot, and the Silhouette Score values, you can identify the optimal number of clusters (K) that best separates the data into meaningful groups.

2.8. K-Means Clustering with Optimal K:

```
model=KMeans(n_clusters=4)
model.fit(telco_data)
print(model.labels_)
```

This code performs K-Means clustering using the chosen optimal K value (in this case, 4).

A KMeans model (`model`) is created with the optimal K value (set to 4 here, but you'll replace this with your chosen value).

The model is fit to the data (`model.fit(telco_data)`). The model's cluster labels (`model.labels_`) are printed. These labels represent the cluster that each data point belongs to.

This concludes the customer segmentation process using K-Means clustering.

3. Feature Selection with Mutual Information (MI)

This code implements feature selection using Mutual Information (MI) to identify the most informative categorical features for predicting customer churn in the telco_data DataFrame.

Explanation:

1: Importing mutual_info_score:

`from sklearn.metrics import mutual_info_score` : This line imports the `mutual_info_score` function from the `sklearn.metrics library` . This function calculates the mutual information between two variables.

2: Selecting Categorical Features:

`categorical_variables = telco_data.drop('Churn', axis=1)` : This line creates a new DataFrame called `categorical_variables` that contains all columns from `telco_data` except the target variable (`'Churn'`). The assumption is that we're only interested in selecting informative features from the categorical columns for churn prediction.

3: Calculating Mutual Information:

`.apply(lambda e: mutual_info_score(e,telco_data.Churn))` : This part applies a lambda function to each column in `categorical_variables`. The lambda function takes a single argument (`e`), which represents a column in the DataFrame. Inside the function: `mutual_info_score(e, telco_data.Churn)` : This calculates the mutual information between the current column (`e`) and the target variable (`telco_data.Churn`). Mutual information measures the mutual dependence between two variables. In this case, it tells us how much information one variable (e.g., a customer's contract type) tells us about another variable (e.g., whether they churn).

4:Sorting Feature Importance:

`.sort_values(ascending=False)` : This sorts the resulting Series (containing the mutual information scores for each feature) by their scores in descending order (`ascending=False`). This ranks the features based on their relevance to the target variable (churn), with higher scores indicating more informative features.

4:Printing Feature Importance:

`print(feature_importance)` : This line prints the Series containing the feature importances (mutual information scores) for all categorical features, ranked from most to least informative for churn prediction.

4. Churn Prediction Models with Logistic Regression and Grid Search

This part demonstrates building and evaluating a Logistic Regression model for predicting customer churn in the `telco_data` DataFrame using Grid Search for hyperparameter tuning.

Steps:

1: Import Libraries:

`import pandas as pd` : Imports pandas library for data manipulation.

`from sklearn.model_selection import train_test_split, GridSearchCV` : Imports functions for splitting data and hyperparameter tuning.

`from sklearn.linear_model import LogisticRegression` : Imports Logistic Regression model class.

`from sklearn.preprocessing import StandardScaler` : Imports StandardScaler for data scaling.

`from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score` : Imports functions for evaluation metrics.

2:Data Preparation:

`X = telco_data.drop('Churn', axis=1)` : Creates a new DataFrame X containing all features except the target variable (`'Churn'`).

`y = telco_data['Churn']` : Separates the target variable (`'Churn'`) into a Series `y` .

`X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)` : Splits the data into training and testing sets using `train_test_split` . Here, 20% of the data is allocated for testing (`test_size=0.2`), and the random state is set for reproducibility (`random_state=42`).

3:Data Scaling:

`scaler = StandardScaler()` : Creates a StandardScaler object.

`X_train_scaled = scaler.fit_transform(X_train)` : Fits the scaler to the training data (`X_train`) to learn the scaling parameters (mean and standard deviation).

`X_test_scaled = scaler.transform(X_test)` : Transforms the testing data (`X_test`) using the fitted scaling parameters from the training data. This ensures both training and testing data have the same scale for better model performance.

4:Logistic Regression Model:

`lr_model = LogisticRegression(max_iter=1000)` : Creates a Logistic Regression model object (`lr_model`) with a maximum of 1000 iterations (`max_iter`).

5:Hyperparameter Tuning with Grid Search:

`param_grid = {...}` : Defines a dictionary (`param_grid`) containing potential values for hyperparameters to be explored by Grid Search. In this case, we're searching for the best values of the following hyperparameters:

`'C'` : Regularization parameter (controls the trade-off between model complexity and fitting the data). `'solver'` : Algorithm used to solve the optimization problem (different solvers might have different performance characteristics). grid_search = GridSearchCV(estimator=lr_model, param_grid=param_grid, cv=5, scoring='accuracy'): Creates a GridSearchCV object (grid_search) to perform the hyperparameter search. estimator: The machine learning model to tune (Logistic Regression model in this case). param_grid: The dictionary containing hyperparameter options. cv: Number of cross-validation folds (set to 5 here). scoring: Evaluation metric to use for selecting the best model (set to 'accuracy' here, but you could use other metrics like F1 score).

6:Training the Model:

`grid_search.fit(X_train_scaled, y_train)` : Fits the GridSearchCV object to the training data (`X_train_scaled and y_train`). This performs training with different hyperparameter combinations and evaluates them using cross-validation based on the chosen scoring metric.

7:Results and Evaluation:

`print("Best Hyperparameters:", grid_search.best_params_)` : Prints the best hyperparameter combination found by Grid Search.

`best_model = grid_search.best_estimator_` : Assigns the best model found by Grid Search to the best_model variable.

`y_pred = best_model.predict(X_test_scaled)` : Makes predictions on the testing data (X_test_scaled) using the best model.

`accuracy = accuracy_score(y_test, y_pred)` : Calculates the accuracy of the model's predictions on the testing data.

5. Multiple Linear Regression vs. Simple Logistic Regression for Churn Prediction

This part compares the performance of a simple Logistic Regression model with a single feature (`tenure`) to a multiple Logistic Regression model with multiple features (`tenure` , `MonthlyCharges` , and `TotalCharges`) for predicting customer churn in the `telco_data` DataFrame.

Steps:

1:Feature Selection:

`selected_features = ['tenure', 'MonthlyCharges', 'TotalCharges']` : Defines a list of features (`selected_features`) to be used in the multiple Logistic Regression model. These features were potentially selected based on feature importance or domain knowledge.

2:Data Preparation (Similar to Section 4):

Prepares the data for training and testing, splitting it with a 20% test size and setting a random state for reproducibility. Simple Logistic Regression:

`simple_lr_model = LogisticRegression(max_iter=1000, solver='saga')` : Creates a simple Logistic Regression model (`simple_lr_model`) with the specified hyperparameters.

`X_train_simple = X_train[['tenure']]` : Selects only the ``'tenure'`` feature from the training data for the simple model.

`X_test_simple = X_test[['tenure']]` : Selects only the ``'tenure'`` feature from the testing data for the simple model.

`simple_lr_model.fit(X_train_simple, y_train)` : Trains the simple model using only the ``'tenure'`` feature and the target variable.

`y_pred_simple = simple_lr_model.predict(X_test_simple)` : Makes predictions on the testing data using the simple model.

4:Evaluation of Simple Model:

Similar to Section 4, calculates and prints the accuracy, precision, recall, and F1 score for the simple Logistic Regression model.

5:Multiple Logistic Regression:

`multiple_lr_model = LogisticRegression(max_iter=1000, solver='saga')` : Creates a multiple Logistic Regression model (`multiple_lr_model`) with the same hyperparameters.

`multiple_lr_model.fit(X_train, y_train)` : Trains the multiple model using all features in X_train and the target variable.

`y_pred_multiple = multiple_lr_model.predict(X_test)` : Makes predictions on the testing data using the multiple model.

6:Evaluation of Multiple Model:

Similar to Section 4, calculates and prints the accuracy, precision, recall, and F1 score for the multiple Logistic Regression model. Comparison and Interpretation:

Prints a comparison between the accuracy scores of both models. Highlights the key difference in interpretability: Simple model: Easier to interpret as it only uses one feature (tenure). Multiple model: Might achieve better performance but can be less interpretable due to the influence of multiple features

6.Unveiling Customer Churn with Neural Networks

This part explores a powerful tool for customer churn prediction: Neural Networks. Here's a breakdown of what it does:

1: Data Preparation:

1: Feature Selection: Similar to previous sections, it separates features (`X`) from the target variable (`y`) in the `telco_data` DataFrame.

2: Splitting Data: The data is divided into training and testing sets using `train_test_split` to train the model and evaluate its performance on unseen data.

3: Scaling Data: A `StandardScaler` ensures all features have similar scales, improving the training process for the neural network.

2: Building the Neural Network:

1: Sequential Model: A `tf.keras.Sequential` model is created, defining the layers the network will have.

2: Hidden Layers:

The first hidden layer has 64 neurons with a ReLU activation function, introducing non-linearity for complex pattern recognition.

A Dropout layer with a 50% dropout rate helps prevent overfitting by randomly dropping out neurons during training.

The second hidden layer has 32 neurons with a ReLU activation for further feature extraction.

3: Output Layer: The final layer has 1 neuron with a sigmoid activation function, suitable for binary classification (churn or not churn).

3: Training and Evaluation:

1: Compiling the Model: The model is compiled, specifying the optimizer (`'adam'`), loss function (`'binary_crossentropy' for binary classification), and metrics (`'accuracy'`).

2: Training the Model: The model is trained on the training data (`X_train_scaled and y_train`) for 20 epochs (iterations) with a batch size of 32. A validation split of 20% is used to monitor performance during training and prevent overfitting.

3: Making Predictions: The trained model predicts churn probabilities (`y_pred_proba`) for the testing data (`X_test_scaled`).

4: Thresholding Predictions: A threshold of 0.5 is applied to convert probabilities into churn classifications (`y_pred`). Customers with a predicted probability above 0.5 are classified as likely to churn.

5: Evaluation Metrics: Similar to previous sections, various metrics (accuracy, precision, recall, F1 score) are calculated to assess the model's performance on the unseen testing data.

Findings Summary:

Customer Segmentation Insights:

Customer segmentation revealed distinct clusters based on behavior and demographics. Different customer segments exhibited varying churn rates and behaviors. Clustering analysis helped identify high-risk customer segments that require targeted retention strategies.

Feature Importance Analysis:

Features such as tenure, monthly charges, and total charges emerged as important predictors of churn. Understanding the impact of these features can help businesses focus their efforts on addressing key factors influencing churn.

Effectiveness of Churn Prediction Models:

Logistic regression models demonstrated reasonable performance in predicting churn, with accuracy above 0.8. The neural network showed competitive performance compared to logistic regression models but required more computational resources. Multiple linear regression models provided insights into the impact of additional features on predictive accuracy and interpretability.

Implications for Reducing Customer Churn:

Implement targeted retention strategies for high-risk customer segments identified through clustering analysis. Focus on improving customer satisfaction and loyalty, particularly among customers with shorter tenure and higher charges. Continuously monitor and analyze customer churn patterns to adapt retention strategies effectively. Utilize predictive models to identify at-risk customers and proactively engage with them to prevent churn. Personalize marketing and communication strategies based on customer segmentation to enhance engagement and satisfaction.

Recommendations for Retaining Customers:

Segment-Specific Retention Strategies:

Tailor retention efforts based on the characteristics and behaviors of different customer segments. Offer personalized incentives or discounts to high-value customers to encourage loyalty.

Enhanced Customer Experience:

Focus on delivering exceptional customer service and support to improve satisfaction levels. Address common pain points and concerns raised by customers through feedback mechanisms.

Proactive Engagement:

Implement proactive outreach initiatives to engage with customers before they reach the churn threshold. Use predictive models to identify early warning signs of potential churn and intervene promptly.

Value-added Services:

Introduce value-added services or features that enhance the overall customer experience and provide additional value to subscribers. Continuous Monitoring and Analysis: Regularly monitor churn metrics and conduct ongoing analysis to identify evolving trends and patterns. Stay updated with customer preferences and market dynamics to adapt retention strategies accordingly.