Analyzing Customer Retention in the Online Food Delivery Business

LAB PROJECT SUBMISSION

Submitted to: Dr. Jhilik Bhattacharya

Submitted by :102103296,102103301,102103302,102103305

Problem statement:

To predict customer churn and help online food delivery companies retain their customers using various input parameters and reviews given by the customers

Description of problem:

The project would analyze the data collected from customers and their interactions with the online food delivery platforms to identify patterns and predict which customers are most likely to stop using the platform's services. The project is about predicting customer churn in the online food delivery industry. It discusses the history of food delivery and how the industry has evolved over time, leading to the growth of online platforms that connect with restaurants. The goal of the project is to predict customer churn and help online food delivery companies retain their customers

Code and Explanation

```
!pip install datacleaner > tempor.txt
```

The command is installing the "datacleaner" package using "pip",which is used for data cleaning or data preprocessing tasks.The output of the installation command will be redirected(typically includes information about the installation progress, success or failure messages, and any errors encountered,used forlater reference or analysis.) and stored in file "tempor.txt"

```
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
import seaborn as sns
plt.style.use('ggplot')
pd.options.display.max_columns = 100
import datacleaner as dc
```

The following lines of code are importing various Python packages:

import pandas as pd: This imports the "pandas" library and gives it the alias "pd". Pandas is a popular data manipulation and analysis library that provides data structures like DataFrame and Series, as well as various data manipulation and analysis functions.

import numpy as np: This imports the "numpy" library and gives it the alias "np". Numpy is a powerful numerical computing library that provides support for working with arrays, matrices, and mathematical functions.

import matplotlib.pylab as plt: This imports the "matplotlib" library and gives access to its "pylab" interface using the alias "plt". Matplotlib is a widely used plotting library in Python that provides various visualization capabilities.

import seaborn as sns: This imports the "seaborn" library and gives it the alias "sns". Seaborn is a statistical data visualization library that is built on top of matplotlib and provides additional high-level plotting functions for statistical analysis.

plt.style.use('ggplot'): This sets the plotting style to "ggplot", which is a popular style for creating visually appealing and informative plots in matplotlib.

pd.options.display.max_columns = 100: This sets the maximum number of columns to be displayed when using pandas for data analysis to 100, which means that pandas will display up to 100 columns in its output.

import datacleaner as dc: This imports the "datacleaner" library and gives it the alias "dc". "datacleaner" is likely a package used for data cleaning or data preprocessing tasks, as mentioned in the previous answer.

```
df = pd.read_csv('onlinedeliverydata.csv')
df.head()
```

pd.read_csv('onlinedeliverydata.csv'): This command reads the data from the CSV file named "onlinedeliverydata.csv" and creates a pandas DataFrame object called "df".

df.head(): This command displays the first few rows (by default, the first 5 rows) of the DataFrame "df".

```
df.describe()
```

	Age	Family size	latitude	longitude	Pin code
count	388.000000	388.000000	388.000000	388.000000	388.000000
mean	24.628866	3.280928	12.972058	77.600160	560040.113402
std	2.975593	1.351025	0.044489	0.051354	31.399609
min	18.000000	1.000000	12.865200	77.484200	560001.000000
25%	23.000000	2.000000	12.936900	77.565275	560010.750000
50%	24.000000	3.000000	12.977000	77.592100	560033.500000
75%	26.000000	4.000000	12.997025	77.630900	560068.000000
max	33.000000	6.000000	13.102000	77.758200	560109.000000

The command df.describe() is used to generate descriptive statistics of the data stored in a pandas DataFrame object called "df" in Python which can be useful for initial exploratory data analysis (EDA)

```
df.isna().sum()
```

df.isna().sum() computes the count of missing or null values for each column in the DataFrame "df". This can be useful for identifying columns with missing data and assessing the completeness of the dataset

```
#Count distinct values of categorical columns
print(df.drop(['Age', 'Family size', 'latitude', 'longitude',
                 'Pin code', 'Reviews'], axis=1).apply(lambda col: col.nunique()))
```

df.drop(['Age', 'Family size', 'latitude', 'longitude', 'Pin code', 'Reviews'], axis=1): This is a pandas DataFrame method that drops the specified columns from the original DataFrame "df".

apply(lambda col: col.nunique()): This is a pandas DataFrame method that applies a function to each column in the resulting DataFrame after dropping the specified columns. The function passed to apply() is a lambda function that computes the number of unique values in each column, using the nunique() method. The syntax computes the count of missing or null values for each column in the DataFrame "df". This can be useful for identifying columns with missing data and assessing the completeness of the dataset.

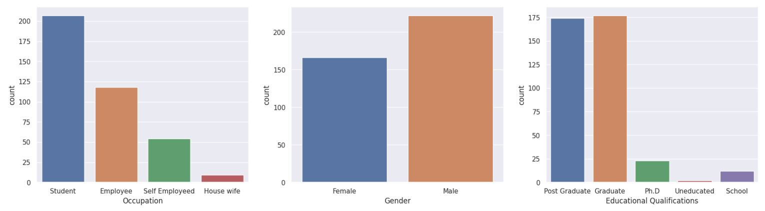
```
df['Output'].value_counts()
```

The command is used to compute and display the count of each unique value in the 'Output' column of a pandas DataFrame object called "df" in Python.

```
# How many people will buy again
#autopct displaying percantages
ax = df['Output'].value_counts().plot(kind='pie', title="How many will buy again",
                                      autopct='%1.1f%%',shadow=True, colors=['lime', 'tomato'], explode=[0.05,0.05])
```

The command creates a pie chart plot of the counts of unique values in the 'Output' column of the DataFrame "df", with specified parameters for the chart type, title, percentage labels, shadow effects, colors, and slice separation. The resulting plot is assigned to the variable "ax" for further use.

```
sns.set(rc={'figure.figsize':(21.7,5.27)})
fig, ax =plt.subplots(1,3)
sns.countplot(x ='Occupation',data=df,ax=ax[0])
sns.countplot(x ='Gender', data = df,ax=ax[1])
sns.countplot(x ='Educational Qualifications',data=df,ax=ax[2])
fig.show()
```



The command sns.set(rc={'figure.figsize':(21.7,5.27)}) sets the figure size of any subsequent plots created using seaborn (imported as sns) to a width of 21.7 inches and a height of 5.27 inches. This can be useful for customizing the size of the plots to suit specific visualizations requirements.

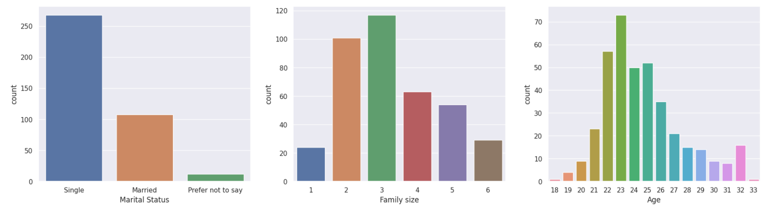
The commands fig, ax =plt.subplots(1,3) create a figure with 1 row and 3 columns, and return the figure object as "fig" and an array of Axes objects as "ax". The "ax" array contains 3 axes objects that represent individual subplots within the figure.

The subsequent commands use seaborn's countplot() function to create count plots for different columns from the DataFrame "df" on the subplots created above:

sns.countplot(x='Occupation', data=df, ax=ax[0]): This creates a count plot of the 'Occupation' column from the DataFrame "df" on the first subplot (ax[0]).

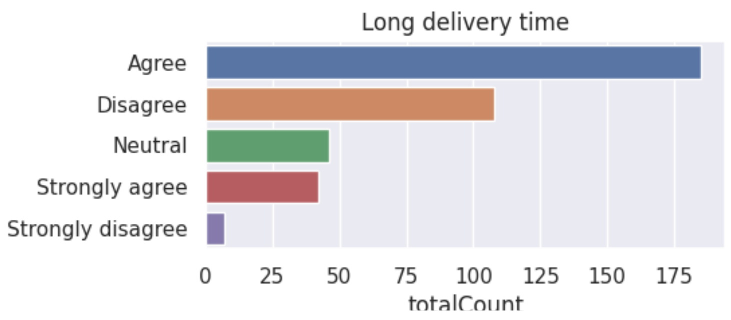
sns.countplot(x='Gender', data=df, ax=ax[1]): This creates a count plot of the 'Gender' column from the DataFrame "df" on the second subplot (ax[1]).etc.

```
sns.set(rc={'figure.figsize':(21.7,5.27)})
fig, ax =plt.subplots(1,3)
sns.countplot(x ='Marital Status',data=df,ax=ax[0])
sns.countplot(x ='Family size', data = df,ax=ax[1])
sns.countplot(x ='Age',data=df,ax=ax[2])
fig.show()
```



expalnation similar to above cell

```
fig, ax =plt.subplots(figsize=(5,2))
z=sns.barplot(x=df["Long delivery time"].value_counts().values, y=df["Long delivery time"].value_counts().index,data=df,
z.set_xlabel('totalCount')
z.set_title("Long delivery time")
fig.show()
```



The command creates a bar plot with the 'Long delivery time' column from the DataFrame "df" on the y-axis and the count of each unique value of 'Long delivery time' on the x-axis. The value_counts() function is used to get the count of each unique value in the 'Long delivery time' column, and the values and index attributes are used to pass the counts and corresponding unique values as the x and y values for the bar plot, respectively. The data parameter specifies the DataFrame to be used for plotting, and the ax parameter specifies the Axes object on which the bar plot should be drawn.

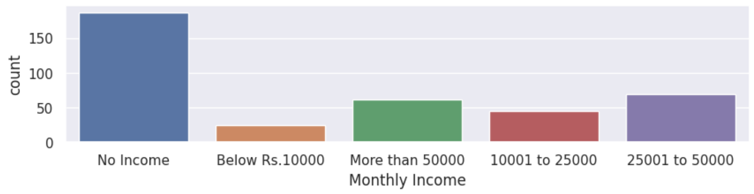
z.set_xlabel('totalCount'): This sets the x-axis label of the bar plot to 'totalCount'.

z.set_title("Long delivery time"): This sets the title of the bar plot to "Long delivery time".

Finally, the command fig.show() displays the figure with the bar plot.

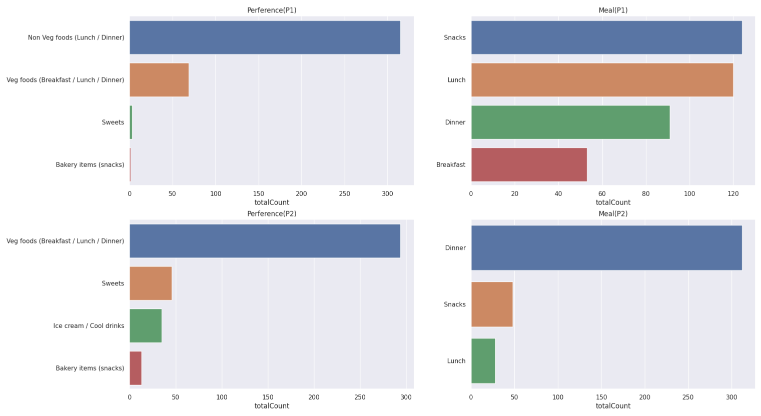
```
fig, ax =plt.subplots(figsize=(10,2))
sns.countplot(x ='Monthly Income',data=df,ax=ax)
```

```
<Axes: xlabel='Monthly Income', ylabel='count'>
```



Same as above

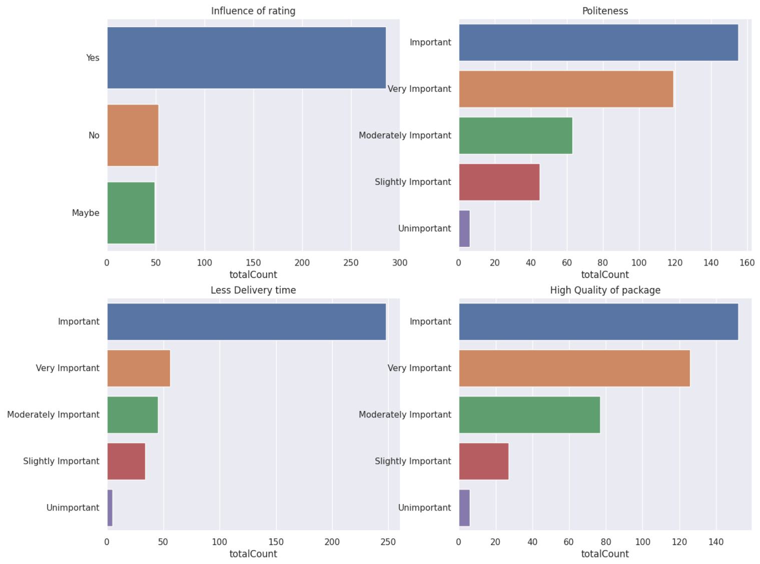
```
fig, ax =plt.subplots(2,2,figsize=(20,12))
z=sns.barplot(x=df["Meal(P1)"].value_counts().values, y=df["Meal(P1)"].value_counts().index,data=df,ax=ax[0][1])
z.set_xlabel('totalCount')
z.set_title("Meal(P1)")
z=sns.barplot(x=df["Meal(P2)"].value_counts().values, y=df["Meal(P2)"].value_counts().index,data=df,ax=ax[1][1])
z.set_xlabel('totalCount')
z.set_title("Meal(P2)")

z.set_xlabel('totalCount')
z=sns.barplot(x=df["Perference(P1)"].value_counts().values, y=df["Perference(P1)"].value_counts().index,data=df,ax=ax[0
z.set_xlabel('totalCount')
z.set_title("Perference(P1)")
z=sns.barplot(x=df["Perference(P2)"].value_counts().values, y=df["Perference(P2)"].value_counts().index,data=df,ax=ax[1

z.set_xlabel('totalCount')
z.set_title("Perference(P2)")
fig.show()
```



Same as above

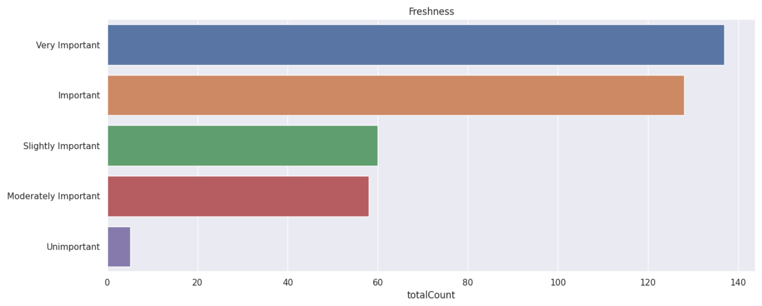
```
fig, ax =plt.subplots(2,2,figsize=(15,12))

z=sns.barplot(x=df["Influence of rating"].value_counts().values, y=df["Influence of rating"].value_counts().index,data=d
z.set_xlabel('totalCount')
z.set_title("Influence of rating")
z=sns.barplot(x=df["Politeness"].value_counts().values, y=df["Politeness"].value_counts().index,data=df,ax=ax[0][1])
z.set_xlabel('totalCount')
```

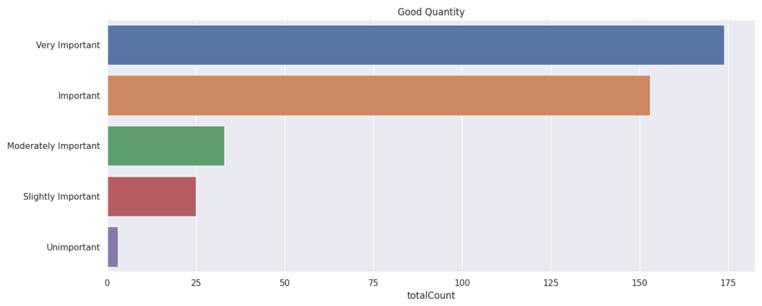
```
z.set_title("Politeness")

z.set_xlabel('totalCount')
z=sns.barplot(x=df["Less Delivery time"].value_counts().values, y=df["Less Delivery time"].value_counts().index,data=df,
z.set_xlabel('totalCount')
z.set_title("Less Delivery time")
z=sns.barplot(x=df["High Quality of package"].value_counts().values, y=df["High Quality of package"].value_counts().inde

z.set_xlabel('totalCount')
z.set_title("High Quality of package")
fig.show()
```

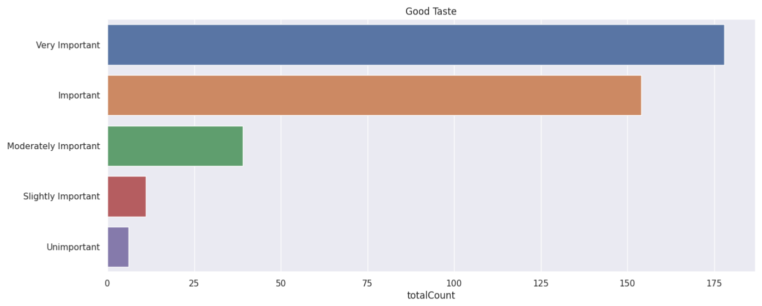


Same as above

```
fig, ax =plt.subplots(3,1,figsize=(15,20))

z=sns.barplot(x=df["Freshness "].value_counts().values, y=df["Freshness "].value_counts().index,data=df,ax=ax[0])
z.set_xlabel('totalCount')
z.set_title("Freshness")
z=sns.barplot(x=df["Good Quantity"].value_counts().values, y=df["Good Quantity"].value_counts().index,data=df,ax=ax[1])
z.set_xlabel('totalCount')
z.set_title("Good Quantity")

z.set_xlabel('totalCount')
z=sns.barplot(x=df["Good Taste "].value_counts().values, y=df["Good Taste "].value_counts().index,data=df,ax=ax[2])
z.set_xlabel('totalCount')
z.set_title("Good Taste")
fig.show()
```







Same as above

Preprocessing

```
cleaned_data = dc.autoclean(df.copy())
cleaned_data.drop(['latitude', 'longitude', 'Pin code', 'Reviews'], axis=1, inplace=True)
cleaned_data.head()
```

	Age	Gender	Marital Status	Occupation	Monthly Income	Educational Qualifications	Family size	Me
0	20	0	2	3	4	2	4	
1	24	0	2	3	2	0	3	
2	22	1	2	3	2	2	3	
3	22	0	2	3	4	0	6	
4	22	1	2	3	2	2	4	

The commands above are using the datacleaner library to perform data cleaning on a DataFrame named df.

cleaned_data = dc.autoclean(df.copy()): This uses the autoclean() function from the datacleaner library to perform automated data cleaning on the DataFrame df. The copy() method is used to create a copy of the original DataFrame before cleaning, so that the original data is not modified. The cleaned data is returned and assigned to a new DataFrame called cleaned_data.

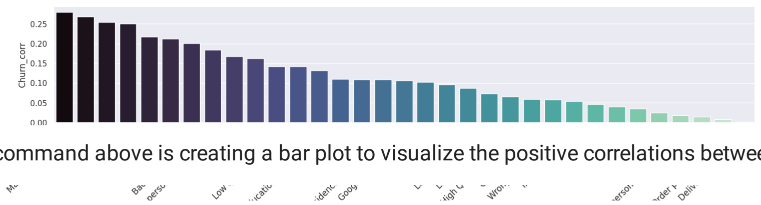
cleaned_data.drop(['latitude', 'longitude', 'Pin code', 'Reviews'], axis=1, inplace=True): This drops the columns 'latitude', 'longitude', 'Pin code', and 'Reviews' from the cleaned_data DataFrame using the drop() function with axis=1 to specify columns and inplace=True to modify the DataFrame in place without creating a new copy.

cleaned_data.head(): This displays the first few rows of the cleaned data using the head() function.

```
colon = cleaned_data.columns[:-1].tolist()
list_corr = []
for i in colon:
  cor = cleaned_data[i].corr(cleaned_data['Output'])
  list_corr.append(cor)
data_corr = pd.DataFrame(list(zip(colon, list_corr)), columns=['Attributes','Churn_corr']).set_index('Attributes')
data_corr_pos = data_corr[data_corr['Churn_corr']>=0].sort_values('Churn_corr', ascending=False)
data_corr_neg = data_corr[data_corr['Churn_corr']<0].sort_values('Churn_corr')
```

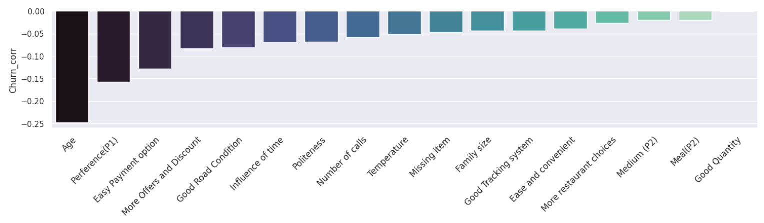
The commands above are performing correlation analysis on the cleaned data stored in the cleaned_data DataFrame.

```
#Plot positive correlations
plt.figure(figsize=(18,3))
sns.barplot(x=data_corr_pos.index, y=data_corr_pos.Churn_corr, data=data_corr, palette="mako")
plt.xlabel(None)
plt.xticks(rotation=45, ha='right', rotation_mode='anchor', fontsize = 12);
```



The command above is creating a bar plot to visualize the positive correlations between the attributes and the target variable ('Output').

```
#Plot negative correlations
plt.figure(figsize=(18,3))
sns.barplot(x=data_corr_neg.index, y=data_corr_neg.Churn_corr, data=data_corr, palette="mako")
plt.xlabel(None)
plt.xticks(rotation=45, ha='right', rotation_mode='anchor', fontsize = 12);
```



The command above is creating a bar plot to visualize the negative correlations between the attributes and the target variable ('Output').

```
dataset = cleaned_data.copy()
```

The command dataset = cleaned_data.copy() creates a copy of the cleaned_data DataFrame and assigns it to a new DataFrame variable called dataset.

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn import metrics
```

All import statements in Python for importing specific modules from the scikit-learn library, which is a popular machine learning library in Python. StandardScaler: This module provides a class for standardizing the features of a dataset, which is a common preprocessing step in machine learning to scale numerical features to have zero mean and unit variance.

train_test_split: This module provides a function for splitting a dataset into training and testing sets, which is commonly used to evaluate the performance of machine learning models.

metrics: This module provides various metrics for evaluating the performance of machine learning models, such as accuracy, precision, recall, F1-score, etc. These metrics are commonly used to assess the quality of predictions made by a machine learning model.

```
X = dataset.drop(["Output"],axis=1)
y = dataset["Output"]

sc = StandardScaler()
X[X.columns[:]] = sc.fit_transform(X[X.columns[:]])
X.head()
```

	Age	Gender	Marital Status	Occupation	Monthly Income	Educational Qualifications	
0	-1.557620	-1.156438	0.657391	0.826756	0.863220	0.899506	
1	-0.211614	-1.156438	0.657391	0.826756	-0.479182	-1.034058	-
2	-0.884617	0.864724	0.657391	0.826756	-0.479182	0.899506	-

X = dataset.drop(["Output"],axis=1): This command creates a new dataframe X by dropping the column "Output" from the original dataset dataset. y = dataset["Output"]: This command creates a new series y containing the values of the "Output" column from the original dataset dataset.

sc = StandardScaler(): This command creates an instance of the StandardScaler class from the scikit-learn library, which will be used to standardize the features of the dataset.

X[X.columns[:]] = sc.fit_transform(X[X.columns[:]]): This command applies the fit_transform method of the StandardScaler object sc to the features (i.e., the columns) of the dataframe X using the fit_transform method.

```
train_x, test_x, train_y, test_y = train_test_split(X, y ,test_size  = 0.25, random_state=0, stratify=dataset.Output)
print("Train dataset shape: {0}, \nTest dataset shape: {1}".format(train_x.shape, test_x.shape))
```

```
    Train dataset shape: (291, 50),
    Test dataset shape: (97, 50)
```

The command train_x, test_x, train_y, test_y = train_test_split(X, y ,test_size = 0.25, random_state=0, stratify=dataset.Output) is used to split the preprocessed dataset X and target variable y into training and testing sets for machine learning. test_size: The proportion of the dataset that should be used for testing. In this case, it is set to 0.25, meaning that 25% of the data will be used for testing and 75% for training.

random_state: The random seed for reproducibility. It ensures that the same random split is used every time the code is run with the same random_state value, which is useful for obtaining consistent results.

stratify: An optional parameter that is set to dataset.Output, which ensures that the class distribution in the target variable y is preserved during the split.

```
from sklearn.neighbors import KNeighborsClassifier
knnmodel = KNeighborsClassifier(n_neighbors=3, p=2) #p=2 represents Euclidean distance, p=1 represents Manhattan Distanc
knnmodel.fit(train_x, train_y)
knn_pred = knnmodel.predict(test_x)
knn_accuracy = metrics.accuracy_score(test_y, knn_pred) * 100
knn_accuracy
list_acc=[]
list_acc.append(knn_accuracy)
```

The code snippet you provided is using the KNeighborsClassifier algorithm from scikit-learn to train a k-nearest neighbors (KNN) classifier for classification tasks.

Here's a brief overview of the steps:

Import the KNeighborsClassifier class from sklearn.neighbors. Create an instance of the KNeighborsClassifier class with specified parameters. In this case, n_neighbors is set to 3, which represents the number of neighbors to consider for classification, and p is set to 2, which represents the Euclidean distance metric. Fit the KNN model to the training data using the fit() method, passing in train_x as the feature matrix and train_y as the target variable. Predict the target values for the test data using the predict() method, passing in test_x as the feature matrix. Calculate the accuracy of the KNN model by comparing the predicted target values with the true target values from the test data, using the accuracy_score() function from sklearn.metrics. Multiply the result by 100 to convert it to a percentage. Store the calculated accuracy in the variable knn_accuracy.Append the accuracy in the final list of accuracies

```
def doknn(row):
  dist = 0
  n = train_x.shape[0]
  ans = [0] * n
  ty = train_y.copy()
  ty = ty.reset_index(drop=True)
  colu = train_x.columns.to_list()
  for r in range(train_x.shape[0]):
    for col in range(train_x.shape[1]):
      t = train_x[colu[col]].values[r] - test_x[colu[col]].values[row]
      ans[r] += t * t
  retval = [(ans[i], ty[i]) for i in range(n)]
  retval = sorted(retval)
  return retval

def findaccKnn(k):
  n = test_x.shape[0]
  ty = test_y.reset_index(drop=True)
  correctans = 0
  for i in range(n):
    dist = doknn(i)
    count1 = 0
    for _ in range(k):
      count1 += dist[_][1]
    if (count1 > k - count1):
      ans = 1
    else:
      ans = 0
    if (ans == ty[i]):
      correctans+=1
  print(correctans * 100 / n)
```

The code snippet you provided defines two functions for performing k-nearest neighbors (KNN) classification manually:

doknn(row): This function calculates the distances between a test instance and all the instances in the training data. It computes the sum of squared differences between the feature values of the test instance and each training instance, and stores the results in a list of tuples. The tuples contain the sum of squared differences as the first element and the corresponding target value from the training data as the second element. The list is then sorted based on the sum of squared differences in ascending order. The function returns the sorted list.

findaccKnn(k): This function takes an integer k as input and calculates the accuracy of the KNN classifier manually using the doknn(row) function. It iterates over all the test instances and calls the doknn(row) function for each instance to get the sorted list of distances. It then calculates the count of positive (1) target values in the top k instances of the sorted list, and compares it with k/2 to determine the predicted target value. If the count of positive target values is greater than k/2, the predicted target value is set to 1, otherwise it is set to 0. Finally, the function calculates and prints the accuracy by comparing the predicted target values with the true target values from the test data.

```
findaccKnn(3)
```

```
    85.56701030927834
```

The findaccKnn(3) function, as per the provided code snippet, calculates and prints the accuracy of the k-nearest neighbors (KNN) classifier using a value of k equal to 3. The accuracy is calculated manually by implementing the KNN algorithm using the doknn(row) function.

```
#LOGISTIC
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
model_log = LogisticRegression()

# Train the model

model_log.fit(train_x, train_y)

# Make predictions on the test set
y_pred = model_log.predict(test_x)

# Calculate accuracy
accuracy = accuracy_score(test_y, y_pred)
print("Accuracy:", accuracy*100)
list_acc.append(accuracy*100)
```

```
    Accuracy: 86.5979381443299
```

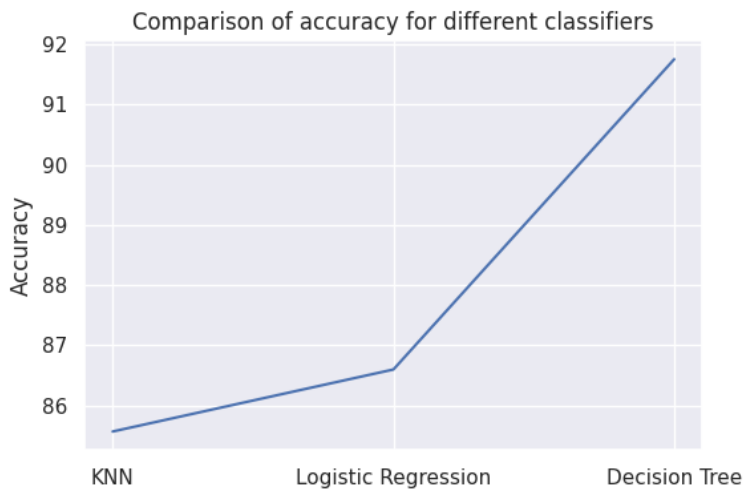
Same as KNN inbuit classifier as used above .Here applies logistic regression for classification to predict the test accuracy

```
from sklearn.tree import DecisionTreeClassifier
model_DT= DecisionTreeClassifier(random_state=0)

# Train the model

model_DT.fit(train_x, train_y)

# Make predictions on the test set
y_pred = model_DT.predict(test_x)

# Calculate accuracy
accuracy = accuracy_score(test_y, y_pred)
print("Accuracy:", accuracy*100)
list_acc.append(accuracy*100)
```

```
    Accuracy: 91.75257731958763
```

Same as KNN inbuit classifier as used above .Here applies Decision tree for classification to predict the test accuracy

```
df1 = pd.DataFrame()
df1[0]=['KNN','Logistic Regression','Decision Tree']
df1[1]=list_acc
fig, ax = plt.subplots(figsize=(6, 4))
d=sns.lineplot(x=0, y=1, data=df1,ax=ax)

plt.title('Comparison of accuracy for different classifiers')
plt.xlabel('classifiers')
plt.ylabel('Accuracy')
plt.show()
```



The provided code snippet creates a DataFrame df1 to store the names of different classifiers ('KNN', 'Logistic Regression', 'Decision Tree') in the first column, and the corresponding accuracy values in the second column. Then, it creates a line plot using Seaborn (sns) library to visualize the comparison of accuracy for different classifiers.

Classification based on Reviews

```
delivery_reviews=df.copy()

#Removing unecessary reviews- Nil value
delivery_reviews= df[~df['Reviews'].isin(['NIL','nil','Nil','No','Nil\n'])]

# Considering only two columns
```

```
delivery_reviews=delivery_reviews[['Reviews','Output']]
```

```
df2=delivery_reviews
```

Creates copy of dataframe df, and remove nil entries .Selects only the 'Reviews' and 'Output' columns from the filtered delivery_reviews DataFrame, and assigns the resulting DataFrame to delivery_reviewsIn summary, the code creates a filtered DataFrame delivery_reviews from the original DataFrame df, removing rows with 'NIL', 'nil', 'Nil', 'No', or 'Nil\n' in the 'Reviews' column, and selecting only the 'Reviews' and 'Output' columns for further analysis.

```
import nltk
nltk.download()
```

download NLTK toolkit

Double-click (or enter) to edit

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df2.Reviews,df2.Output)
```

split the reviews using train test split

```
import string
from sklearn.feature_extraction.text import CountVectorizer

v = CountVectorizer(stop_words='english', lowercase=True, tokenizer=lambda text: text.translate(str.maketrans('', '', s
```

```
X_train_count = v.fit_transform(X_train)
#X_train_count
X_train_count.toarray()[:2]
```

The CountVectorizer class is used to convert a collection of text documents into a matrix of token (word) counts, where each row represents a document and each column represents a unique word in the entire collection of documents.

The CountVectorizer is initialized with the following parameters:

stop_words='english': Specifies that English stopwords should be removed from the text data during tokenization. lowercase=True: Specifies that all text data should be converted to lowercase during tokenization.

tokenizer=lambda text: text.translate(str.maketrans('', '', string.punctuation)).split(): Specifies a custom tokenizer function that removes punctuation from the text data using str.translate() method with string.punctuation as the translation table, and then splits the text into tokens using str.split() method.

The fit_transform() method is then called on the CountVectorizer object v with the X_train data, which is a collection of text documents. This method performs the tokenization, stopword removal, lowercase conversion, and word counting, and returns a sparse matrix representation of the token counts in X_train.

The toarray() method is then called on the resulting sparse matrix to convert it into a dense numpy array, and the first two rows of the resulting array are printed using slicing [:2].

```
from sklearn.naive_bayes import MultinomialNB
model = MultinomialNB()
model.fit(X_train_count,y_train)
```

The MultinomialNB class is a implementation of Naive Bayes algorithm specifically designed for discrete features, such as word counts, which makes it suitable for text classification tasks where the features are typically represented as word counts or other discrete representations.

The fit() method is then called on the MultinomialNB object model with X_train_count as the feature matrix and y_train as the target vector. This method trains the Naive Bayes classifier on the provided training data, which includes the token counts obtained from X_train_count and the corresponding target labels from y_train.

```
X_test_count = v.transform(X_test)
model.score(X_test_count, y_test)
```

```
0.9508196721311475
```

The provided code is using the transform() method on the CountVectorizer object v to transform the test data X_test into token counts using the vocabulary learned from the training data. The resulting token counts are stored in X_test_count.

Then, the score() method is called on the trained MultinomialNB object model with X_test_count as the feature matrix and y_test as the target vector. This method calculates the accuracy of the model on the provided test data, which includes the token counts obtained from X_test_count and the corresponding target labels from y_test. The accuracy score is returned as the output.

✓ 0s completed at 23:37