DSC 550

Inman, Gracie

Term Project Final

11/18/23

Project Goal

Throughout this project I want to be able to predict turn for customers for the Telco Company. From Kaggle, I obtained a dataset that shares Telco customer information and whether or not the customer churned. My goal is to see if there is a significant pattern in those who stay with the company and those who decide to leave and be able to apply that model to future and current customers. Applying it to current customers can provide insight to churn and hopefully allow for intervention before churn happens. Applying it to future customers can hopefully make a lifelong customer. Churn is a large business issue, because if you can identify a churn pattern you have more knowledge in how to prevent it. While significantly less people have turned in this dataset then stayed, roughly 3/10 customers have unsubscribed. It is important to figure out why to help improve retention. Customers are arguably the most important part of a business as they are the revenue that keeps the company open. I plan to look at how each factor influences churn individually and then together to identify key patterns in churn prediction. Looking at the factors individually will allow me to see if one variable is more significant in predicting turn that another variable. I am also interested in how dependents affect turn. I have a theory that with dependents money could possibly be tighter and cause churn in regards to the cost. I am also curious if it is significant enough to consider dependents as a major variable in churn prediction.

Dataset obtained from Kaggle: https://www.kaggle.com/datasets/reyhanarighy/data-telco-customer-churn/

Load Data

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
telco_df = pd.read_csv('data_telco_customer_churn.csv')
telco_df.head()
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Out[2]:

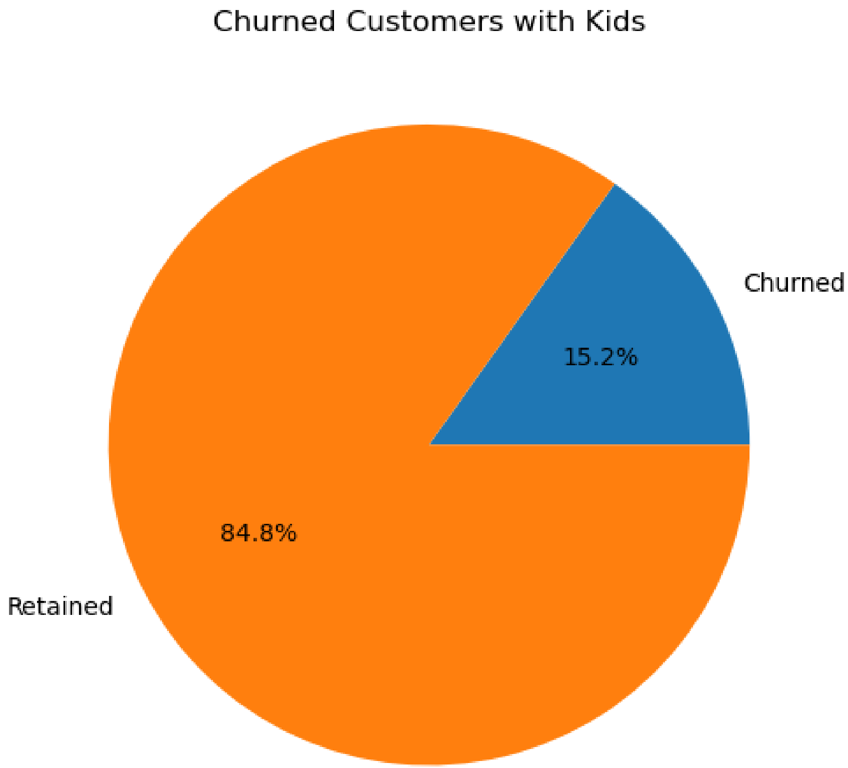
	Dependents	tenure	OnlineSecurity	OnlineBackup	InternetService	DeviceProtection	TechSupport	Contract	PaperlessBilling	Mc
0	Yes	9	No	No	DSL	Yes	Yes	Month-to-month	Yes	
1	No	14	No	Yes	Fiber optic	Yes	No	Month-to-month	Yes	
2	No	64	Yes	No	DSL	Yes	Yes	Two year	No	
3	No	72	Yes	Yes	DSL	Yes	Yes	Two year	No	
4	No	3	No internet service	No internet service	No	No internet service	No internet service	Month-to-month	Yes	

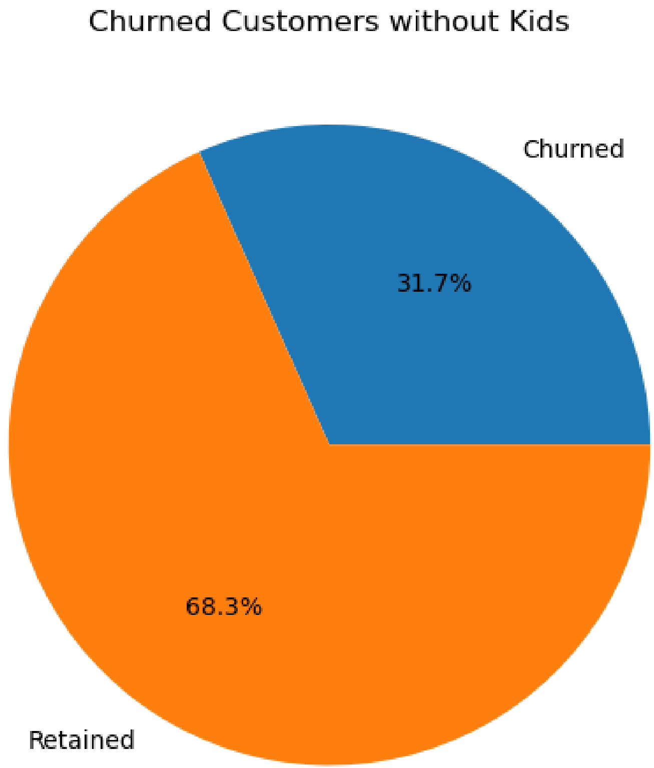
Pie Charts Dependents vs Churn

In [3]:

```
# Filter the DataFrame
churned_with_kids = telco_df[(telco_df['Dependents'] == 'Yes') & (telco_df['Churn'] == 'Yes')].shape[0]
churned_without_kids = telco_df[(telco_df['Dependents'] == 'No') & (telco_df['Churn'] == 'Yes')].shape[0]

# Labels for the two categories
labels = ['With Kids', 'Without Kids']

# Data for the two categories
sizes = [churned_with_kids, churned_without_kids]

# CPie chart for people with kids who churned
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.pie([churned_with_kids, telco_df[(telco_df['Dependents'] == 'Yes') & (telco_df['Churn'] == 'No')].shape[0
        labels=['Churned', 'Retained'], autopct='%1.1f%%')
plt.title('Churned Customers with Kids')

# Pie chart for people without kids who churned
plt.subplot(1, 2, 2)
plt.pie([churned_without_kids, telco_df[(telco_df['Dependents'] == 'No') & (telco_df['Churn'] == 'No')].shape
        labels=['Churned', 'Retained'], autopct='%1.1f%%')
plt.title('Churned Customers without Kids')

# Display the pie charts
plt.tight_layout()
plt.show()
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

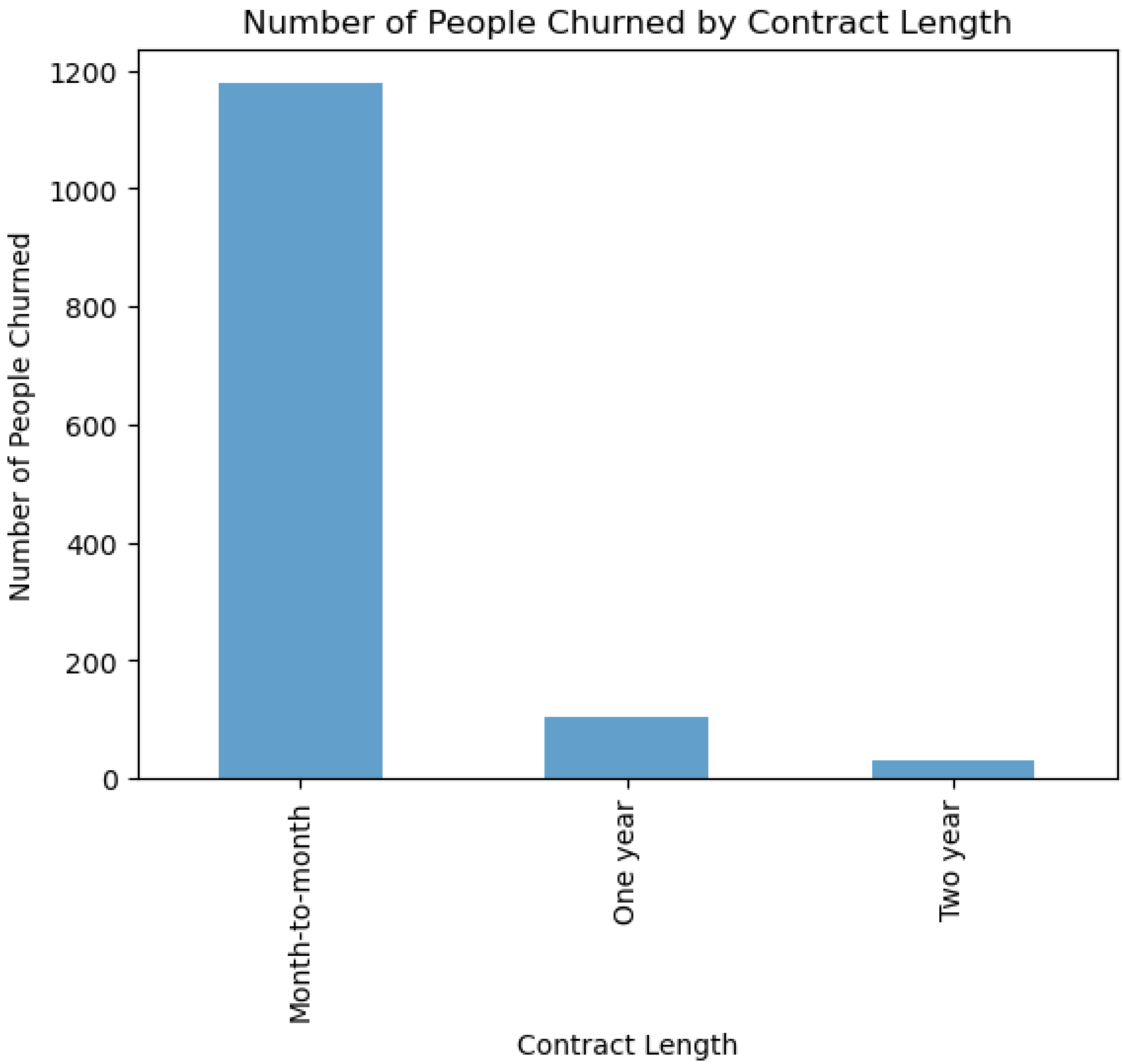




Bar Graph of Contract Terms vs Churn

In [4]:
```
# Group the data
churn_counts = telco_df[telco_df['Churn'] == 'Yes'].groupby('Contract').size()

# Create the bar graph
churn_counts.plot(kind='bar', alpha=0.7)
plt.xlabel('Contract Length')
plt.ylabel('Number of People Churned')
plt.title('Number of People Churned by Contract Length')
plt.show()
```

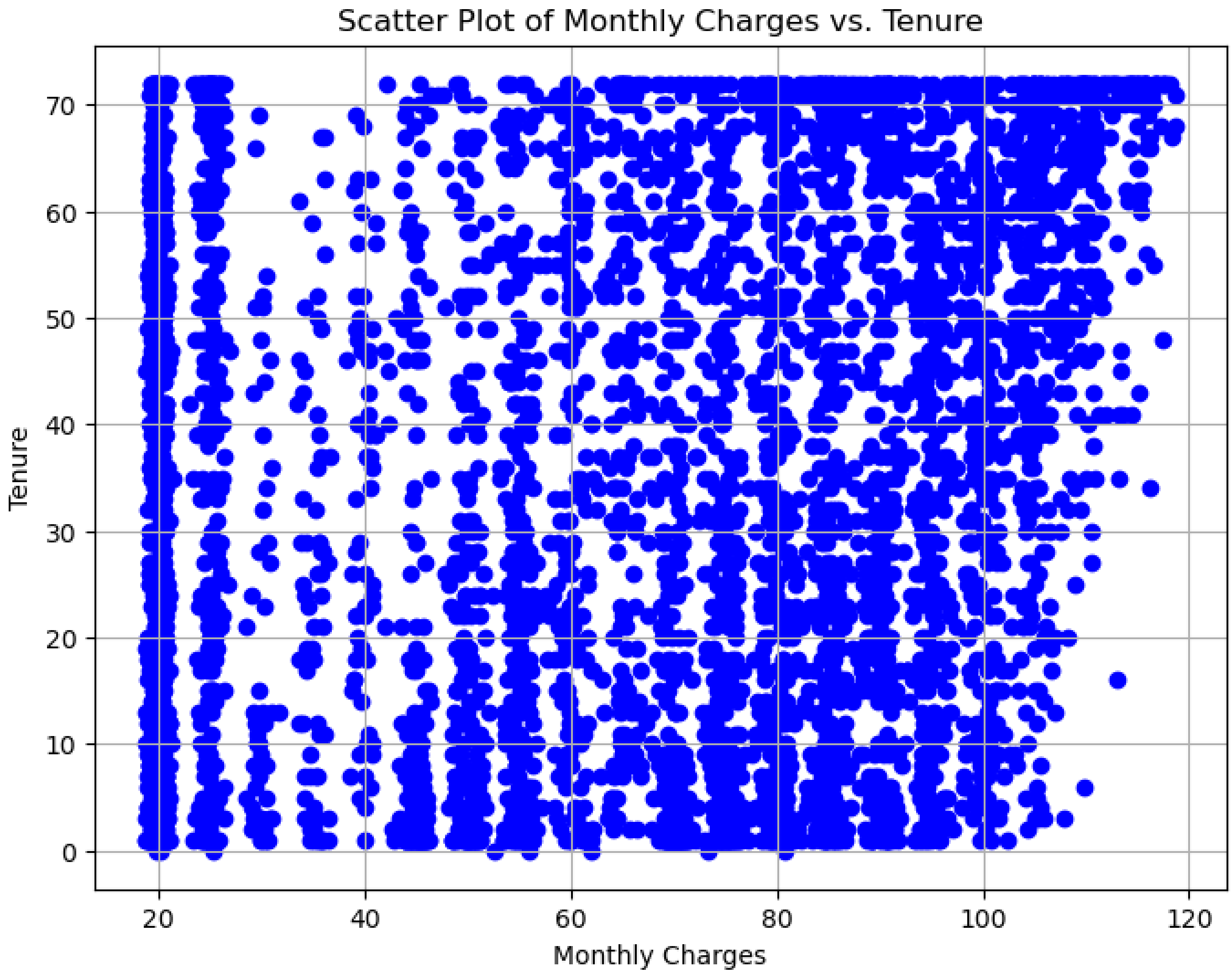
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js



Scatter Plot of Monthly Charges vs Tenure

In [5]:
```
plt.figure(figsize=(8, 6))
plt.scatter(telco_df['MonthlyCharges'], telco_df['tenure'], marker='o', color='b')
plt.xlabel('Monthly Charges')
plt.ylabel('Tenure')
plt.title('Scatter Plot of Monthly Charges vs. Tenure')
plt.grid(True)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

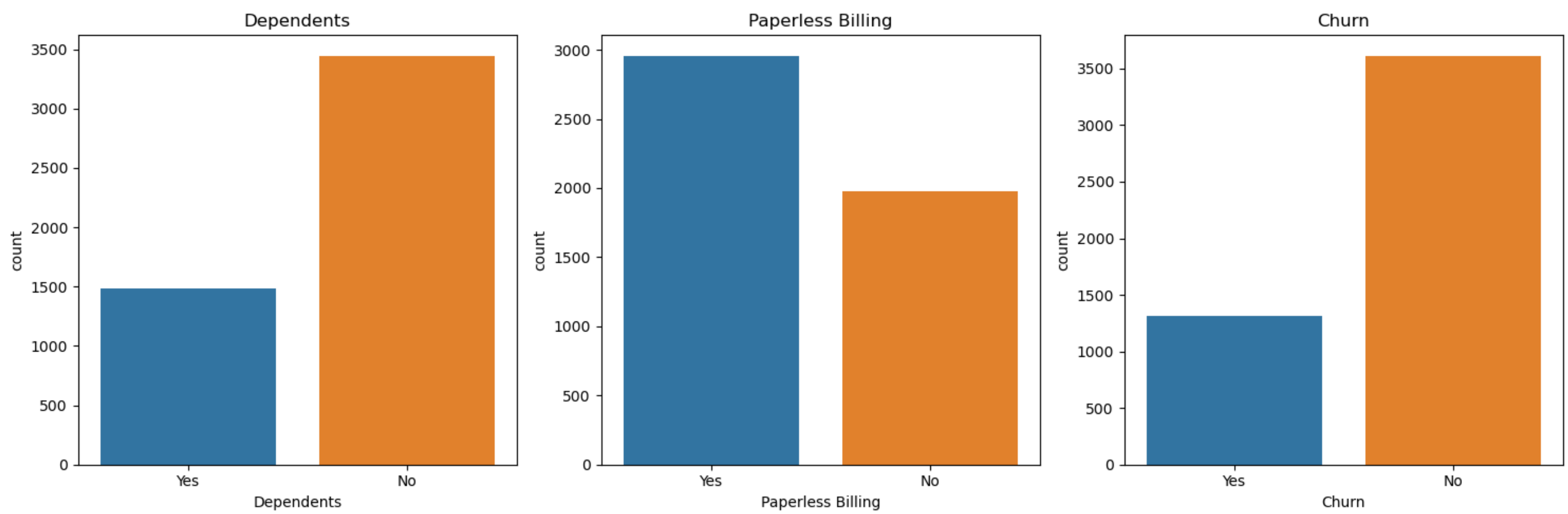


Count Plots

In [6]:
```
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
sns.countplot(x='Dependents', data=telco_df)
plt.xlabel('Dependents')
plt.title('Dependents')

plt.subplot(1, 3, 2)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
sns.countplot(x='PaperlessBilling', data=telco_df)
plt.xlabel('Paperless Billing')
plt.title('Paperless Billing')

plt.subplot(1, 3, 3)
sns.countplot(x='Churn', data=telco_df)
plt.xlabel('Churn')
plt.title('Churn')

plt.tight_layout()
plt.show()
```



Looking at the pie charts, it is shown that a larger percentage of people that churned did not have children. From the bar graph, it was shown that a vast majority of the people that have turned were on a month to month contract. The scatter plot shows no correlation between cost monthly and time with the company. The count plots show that a larger part of the customers do not have dependents. There are also a large amount of customers enrolled in paperless billing, but still a substainal amount that are not enrolled. There is also a significantly larger amount of customers that do not turn.

Milestone 2

In [7]:
```
# Check for missing values
missing_values = telco_df.isnull().sum()
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
print("Missing values in the entire DataFrame:")
print(missing_values)
```

```
Missing values in the entire DataFrame:
Dependents          0
tenure              0
OnlineSecurity      0
OnlineBackup        0
InternetService     0
DeviceProtection    0
TechSupport         0
Contract            0
PaperlessBilling    0
MonthlyCharges      0
Churn               0
dtype: int64
```

Checking for missing values insures that no errors will arise due to missing data.

In [8]:
```
telco_df.head()
```

Out[8]:

	Dependents	tenure	OnlineSecurity	OnlineBackup	InternetService	DeviceProtection	TechSupport	Contract	PaperlessBilling	Mo
0	Yes	9	No	No	DSL	Yes	Yes	Month-to-month	Yes	
1	No	14	No	Yes	Fiber optic	Yes	No	Month-to-month	Yes	
2	No	64	Yes	No	DSL	Yes	Yes	Two year	No	
3	No	72	Yes	Yes	DSL	Yes	Yes	Two year	No	
4	No	3	No internet service	No internet service	No	No internet service	No internet service	Month-to-month	Yes	

Frequently looking at the dataframe allows me to visualize what needs to be done and how the data is transforming.

In [9]:
```
# Remove usless columns
columns_to_remove = ['OnlineSecurity', 'OnlineBackup', 'DeviceProtection', 'TechSupport']
telco_df.drop(columns=columns_to_remove, inplace=True)
telco_df.head()
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Out[9]:

	Dependents	tenure	InternetService	Contract	PaperlessBilling	MonthlyCharges	Churn
0	Yes	9	DSL	Month-to-month	Yes	72.90	Yes
1	No	14	Fiber optic	Month-to-month	Yes	82.65	No
2	No	64	DSL	Two year	No	47.85	Yes
3	No	72	DSL	Two year	No	69.65	No
4	No	3	No	Month-to-month	Yes	23.60	No

Removing columns I will not use will help the code run more smoothly and allows me visualize my data and better.

In [10]:
```
# Checking unique values
unique_value_counts = telco_df.nunique()
print(unique_value_counts)
```

```
Dependents            2
tenure               73
InternetService       3
Contract              3
PaperlessBilling      2
MonthlyCharges     1422
Churn                 2
dtype: int64
```

Checking to make sure that there is the correct number of unique variables in each column before I begin coding the categorical variables.

In [11]:
```
# Standardizing values
category_mapping = {'Month-to-month': 0, 'One year': 1, 'Two year': 2}
telco_df['Contract'] = telco_df['Contract'].map(category_mapping)
telco_df.head()
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Out[11]:

	Dependents	tenure	InternetService	Contract	PaperlessBilling	MonthlyCharges	Churn
0	Yes	9	DSL	0	Yes	72.90	Yes
1	No	14	Fiber optic	0	Yes	82.65	No
2	No	64	DSL	2	No	47.85	Yes
3	No	72	DSL	2	No	69.65	No
4	No	3	No	0	Yes	23.60	No

In [12]:

```
category_mapping1 = {'DSL': 0, 'Fiber optic': 1, 'No': 2}
telco_df['InternetService'] = telco_df['InternetService'].map(category_mapping1)
telco_df.head()
```

Out[12]:

	Dependents	tenure	InternetService	Contract	PaperlessBilling	MonthlyCharges	Churn
0	Yes	9	0	0	Yes	72.90	Yes
1	No	14	1	0	Yes	82.65	No
2	No	64	0	2	No	47.85	Yes
3	No	72	0	2	No	69.65	No
4	No	3	2	0	Yes	23.60	No

In [13]:

```
columns_to_replace = ['Dependents', 'PaperlessBilling', 'Churn']
replace_map = {'Yes': 1, 'No': 0}
telco_df[columns_to_replace] = telco_df[columns_to_replace].replace(replace_map)
telco_df.head()
```

Out[13]:

	Dependents	tenure	InternetService	Contract	PaperlessBilling	MonthlyCharges	Churn
0	1	9	0	0	1	72.90	1
1	0	14	1	0	1	82.65	0
2	0	64	0	2	0	47.85	1
3	0	72	0	2	0	69.65	0
4	0	3	2	0	1	23.60	0

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Changing the categorical values to numbers allow for further analysis. This also helps standardize the data and allow for better visualizations.

In [14]:
```
# Change the column names
new_column_names = {
    'tenure': 'Tenure',
    'InternetService': 'Internet Service',
    'PaperlessBilling': 'Paperless Billing',
    'MonthlyCharges': 'Monthly Charges'
}

telco_df.rename(columns=new_column_names, inplace=True)
telco_df.head()
```

Out[14]:

	Dependents	Tenure	Internet Service	Contract	Paperless Billing	Monthly Charges	Churn
0	1	9	0	0	1	72.90	1
1	0	14	1	0	1	82.65	0
2	0	64	0	2	0	47.85	1
3	0	72	0	2	0	69.65	0
4	0	3	2	0	1	23.60	0

I was getting tripped up on the formating of the column names so I fixed them to standardize the naming and improve readability.

In [15]:
```
# Overview statisitics
telco_df.describe()
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Out[15]:

	Dependents	Tenure	Internet Service	Contract	Paperless Billing	Monthly Charges	Churn
count	4930.000000	4930.000000	4930.000000	4930.000000	4930.000000	4930.000000	4930.000000
mean	0.301014	32.401217	0.867343	0.682759	0.599797	64.883032	0.266937
std	0.458745	24.501193	0.736168	0.828317	0.489989	29.923960	0.442404
min	0.000000	0.000000	0.000000	0.000000	0.000000	18.800000	0.000000
25%	0.000000	9.000000	0.000000	0.000000	0.000000	37.050000	0.000000
50%	0.000000	29.000000	1.000000	0.000000	1.000000	70.350000	0.000000
75%	1.000000	55.000000	1.000000	1.000000	1.000000	89.850000	1.000000
max	1.000000	72.000000	2.000000	2.000000	1.000000	118.650000	1.000000

This shows a brief overview of summary statistics for each column and allows me to look over the data and identify possible issues and next steps.

In [16]:

```
# Check for Outliers
# Calculate Z-scores
telco_df['Tenure_Z'] = (telco_df['Tenure'] - telco_df['Tenure'].mean()) / telco_df['Tenure'].std()
telco_df['Monthly_Charges_Z'] = (telco_df['Monthly Charges'] -
                                 telco_df['Monthly Charges'].mean()) / telco_df['Monthly Charges'].std()
z_score_threshold = 2

# Identify rows with outliers
outliers_tenure = telco_df[abs(telco_df['Tenure_Z']) > z_score_threshold]
outliers_monthly_charges = telco_df[abs(telco_df['Monthly_Charges_Z']) > z_score_threshold]

# Remove the Z-score columns
telco_df.drop(['Tenure_Z', 'Monthly_Charges_Z'], axis=1, inplace=True)

# Indicate if outliers were found
if not outliers_tenure.empty:
    print("Outliers in Tenure column:")
    print(outliers_tenure)
else:
    print("No outliers found in Tenure column.")

if not outliers_monthly_charges.empty:
    print("Outliers in Monthly Charges column:")
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
else:
    print("No outliers found in Monthly Charges column.")
```

```
No outliers found in Tenure column.
No outliers found in Monthly Charges column.
```

Checking for outliers ensures that there will be no bias from outliers later in the data analysis process. There were no outliers present for a threshold of 2 or 3 and therefore no replacement or removal is needed.

In [17]:
```
# Check the shape
telco_df.shape
```

Out[17]:
```
(4930, 7)
```

In [18]:
```
# Remove duplicates
telco_df = telco_df.drop_duplicates()
telco_df.shape
```

Out[18]:
```
(4830, 7)
```

Removing duplicates eliminates bias from having more than one of the same data point. A hundred data points were duplicates and were removed.

In [19]:
```
# Performing SMOTE as suggested to deal with imbalanced data.
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split

x = telco_df.drop(columns='Churn')
y = telco_df['Churn']

# Split the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# SMOTE
smote = SMOTE(sampling_strategy='auto', random_state=42)

# Apply SMOTE to the training data
x_resampled, y_resampled = smote.fit_resample(x_train, y_train)
```

In [20]:
```
resampled_data = pd.concat([x_resampled, y_resampled], axis=1)
resampled_data.reset_index(drop=True, inplace=True)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Out[20]:

	Dependents	Tenure	Internet Service	Contract	Paperless Billing	Monthly Charges	Churn
0	0	46	0	1	1	69.10	0
1	0	56	2	2	0	19.70	0
2	1	24	2	2	1	19.70	0
3	0	51	1	1	0	87.55	0
4	0	8	0	0	1	66.70	0

In [21]:
```
resampled_data.shape
```

Out[21]:
```
(5734, 7)
```

SMOTE was used due to the dataset being highly imbalanced. An imbalanced dataset can cause innacurate assumptions due to the higher presence of one varaiable.

Milestone 3

The data was already loaded, cleaned and split previously. For this model, we want to evaluate the model based on accuracy and precision. This is because we want to predict with the most accuracy and with the most precision if a customer is going to turn so we can target the customer to prevent churn. Due to the large amount of categorical variables, my first model will be logisitic regression.

In [22]:
```
# load libraries
from sklearn.metrics import accuracy_score, classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import precision_score
```

Logistic Regression

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

In [23]:
```
# train the model on the training data
model = LogisticRegression()
model.fit(x_train, y_train)
```

Out[23]:
```
▾ LogisticRegression
LogisticRegression()
```

In [24]:
```
# Evaluate the model using the test data.
y_pred = model.predict(x_test)
accuracy1 = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy1)
```

```
Accuracy: 0.7763975155279503
```

In [25]:
```
# Make predictions on the test set
y_pred = model.predict(x_test)

# Calculate precision
precision = precision_score(y_test, y_pred)

print("Precision:", precision)
```

```
Precision: 0.6567164179104478
```

Random Forest

The next model I would like to try is Random Forest. This is due to how random forest does not assume a linear relationship and can help prevent overfitting.

In [26]:
```
# Train the model using the training data
random_forest = RandomForestClassifier(n_estimators=100, random_state=42)
random_forest.fit(x_train, y_train)
```

Out[26]:
```
▾ RandomForestClassifier
RandomForestClassifier(random_state=42)
```

In [27]:
```
# Evaluate the model using the test data
y_pred = random_forest.predict(x_test)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
# Calculate accuracy
accuracy2 = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy2)

# Generate a classification report
report = classification_report(y_test, y_pred)
print("Classification Report:")
print(report)
```

```
Accuracy: 0.7587991718426501
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.87      0.84       687
           1       0.60      0.48      0.53       279

    accuracy                           0.76       966
   macro avg       0.70      0.68      0.69       966
weighted avg       0.75      0.76      0.75       966
```

Naive Bayes

Next, I would like to see how Naive Bayes performs with the data. It can work well with a smaller amount of data.

In [28]:
```
# Train the data using the training set
naive_bayes = GaussianNB()
naive_bayes.fit(x_train, y_train)
```

Out[28]:
```
▾ GaussianNB
GaussianNB()
```

In [29]:
```
# Make predictions on the test data
y_pred = naive_bayes.predict(x_test)

# Calculate accuracy
accuracy3 = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy3)

# Generate a classification report
st, y_pred)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
print("Classification Report:")
print(report)
```

```
Accuracy: 0.7577639751552795
Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.76      0.82       687
           1       0.56      0.74      0.64       279

    accuracy                           0.76       966
   macro avg       0.72      0.75      0.73       966
weighted avg       0.79      0.76      0.77       966
```

Decision Tree

Decision Trees are commonly used in turn prediction. This is common because it splits the data into smaller groups and considers numerical and categorical variable.

In [30]:
```
# Train the Desicion Tree model using the training data.
decision_tree = DecisionTreeClassifier(random_state=42)
decision_tree.fit(x_train, y_train)
```

Out[30]:
```
▾ DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

In [31]:
```
# Make predictions using the test data
y_pred = decision_tree.predict(x_test)

# Calculate accuracy
accuracy4 = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy4)

# Generate a classification report
report = classification_report(y_test, y_pred)
print("Classification Report:")
print(report)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
Accuracy: 0.7184265010351967
Classification Report:
              precision    recall  f1-score   support

           0       0.79      0.82      0.80       687
           1       0.51      0.48      0.49       279

    accuracy                           0.72       966
   macro avg       0.65      0.65      0.65       966
weighted avg       0.71      0.72      0.72       966
```

In [32]:
```
print('Accuracy and Precision Summary:')
print(f'Logistic Regression Accuracy: {accuracy1:.2f}\n Precision: 0.66')
print(f'Random Forest Accuracy: {accuracy2:.2f}\n Precision: 0.60')
print(f'Naive Bayes Accuracy: {accuracy3:.2f}\n Precision: 0.56')
print(f'Decision Tree Accuracy: {accuracy4:.2f}\n Precision: 0.51')
```

```
Accuracy and Precision Summary:
Logistic Regression Accuracy: 0.78
 Precision: 0.66
Random Forest Accuracy: 0.76
 Precision: 0.60
Naive Bayes Accuracy: 0.76
 Precision: 0.56
Decision Tree Accuracy: 0.72
 Precision: 0.51
```

Based on accuracy, the best model is logisitic regression which has the highest accuracy. Random forest and naive bayes have similar accuracies with 0.759 and 0.758 retrospectively. Decision tree had the lowest accuracy at 0.718. The logistic regression also has the highest precision of accurately predicting if the customer will turn. This is importantant because the model can be accurate for predicting if a customer will not turn but very innacurate when determining if a customer will turn. For my analysis, the logisitic regression model is the best fit.

In [33]:
```
# Cross validation of Accuracy

from sklearn.model_selection import cross_val_score, KFold
from sklearn.linear_model import LogisticRegression

# Create a logistic regression model
model = LogisticRegression()

k = 10
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
# Create a KFold cross-validation object
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Perform cross-validation and get accuracy scores
accuracy_scores = cross_val_score(model, x, y, cv=kf, scoring='accuracy')

# Print the accuracy scores for each fold
for i, score in enumerate(accuracy_scores):
    print(f'Fold {i+1} - Accuracy: {score:.2f}')

# Calculate and print the mean accuracy score
mean_accuracy = accuracy_scores.mean()
print(f'Mean Accuracy: {mean_accuracy:.2f}')
```

```
Fold 1 - Accuracy: 0.79
Fold 2 - Accuracy: 0.76
Fold 3 - Accuracy: 0.78
Fold 4 - Accuracy: 0.78
Fold 5 - Accuracy: 0.78
Fold 6 - Accuracy: 0.82
Fold 7 - Accuracy: 0.78
Fold 8 - Accuracy: 0.79
Fold 9 - Accuracy: 0.78
Fold 10 - Accuracy: 0.82
Mean Accuracy: 0.79
```

In [34]:

```
# Percision Cross Validation
model = LogisticRegression()

# Define the number of folds
k = 10

# Create a KFold cross-validation object
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Perform cross validation and get precision scores
precision_scores = cross_val_score(model, x, y, cv=kf, scoring='precision')

# Print the precision scores for each fold
for i, score in enumerate(precision_scores):
    print(f'Fold {i+1} - Precision: {score:.2f}')

# Calculate and print the mean precision
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
mean_precision = precision_scores.mean()
print(f'Mean Precision: {mean_precision:.2f}')
```

```
Fold 1 - Precision: 0.68
Fold 2 - Precision: 0.61
Fold 3 - Precision: 0.65
Fold 4 - Precision: 0.67
Fold 5 - Precision: 0.52
Fold 6 - Precision: 0.61
Fold 7 - Precision: 0.62
Fold 8 - Precision: 0.65
Fold 9 - Precision: 0.64
Fold 10 - Precision: 0.60
Mean Precision: 0.63
```

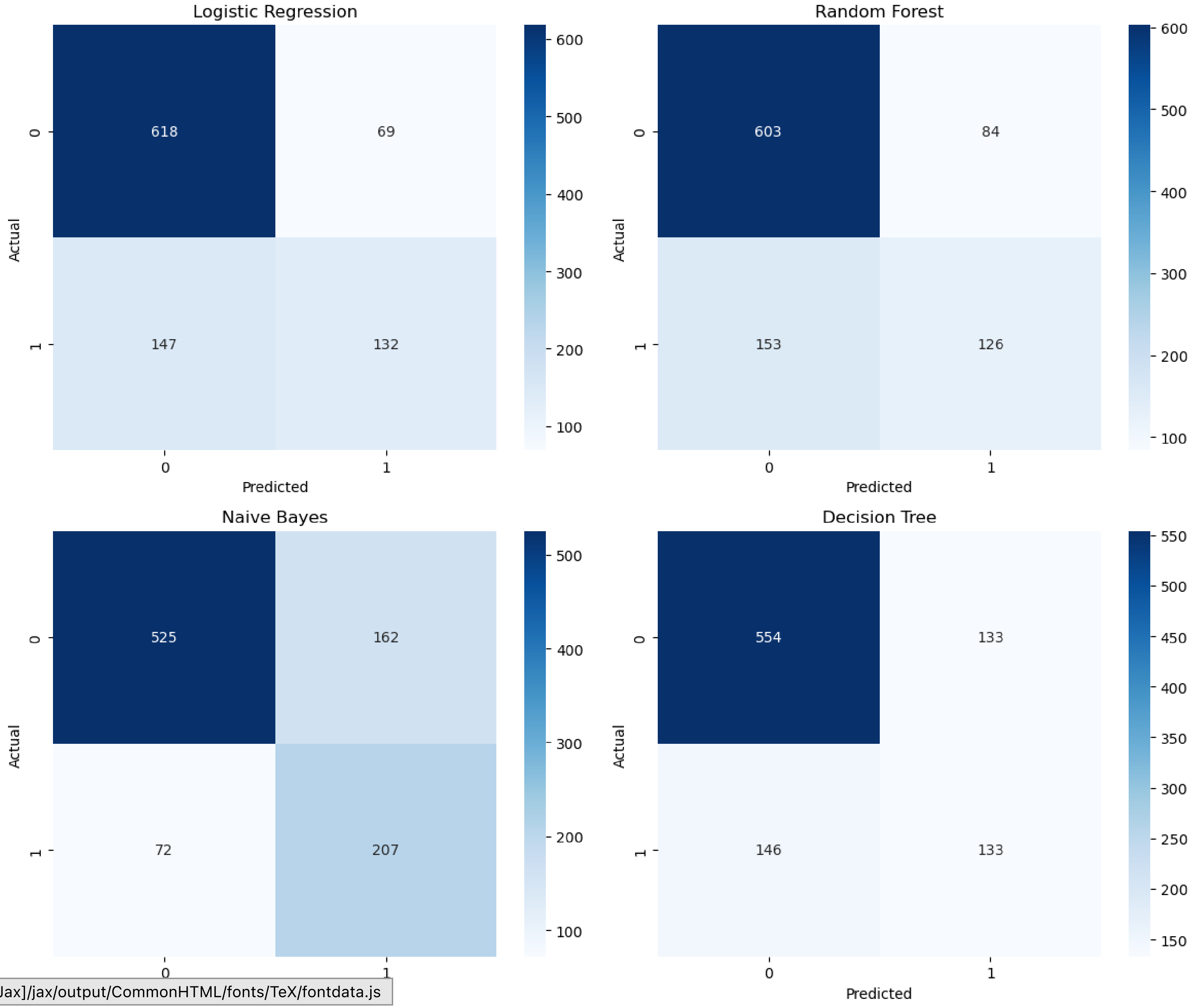
Accuracy obtained from the model was 0.78 with a cross validation accuracy of 0.79. The percision was calculated as 0.66 and the cross validation was 0.63. Neither difference is substancial. Cross validation indicates model is performing well and consistently. This indicates that the model will perform well with unseen data.

In [ ]:
```
# Adding Confusion Matrix
```

In [40]:
```
from sklearn.metrics import confusion_matrix
# Logistic Regression
logistic_regression_model = LogisticRegression()
logistic_regression_model.fit(x_train, y_train)
y_pred_lr = logistic_regression_model.predict(x_test)

# Random Forest
random_forest_model = RandomForestClassifier()
random_forest_model.fit(x_train, y_train)
y_pred_rf = random_forest_model.predict(x_test)

# Naive Bayes
naive_bayes_model = GaussianNB()
naive_bayes_model.fit(x_train, y_train)
y_pred_nb = naive_bayes_model.predict(x_test)

# Decision Tree
decision_tree_model = DecisionTreeClassifier()
decision_tree_model.fit(x_train, y_train)
y_pred_dt = decision_tree_model.predict(x_test)

cm_lr = confusion_matrix(y_test, y_pred_lr)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
cm_rf = confusion_matrix(y_test, y_pred_rf)
cm_nb = confusion_matrix(y_test, y_pred_nb)
cm_dt = confusion_matrix(y_test, y_pred_dt)

# Plotting confusion matrices
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Logistic Regression
sns.heatmap(cm_lr, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1], yticklabels=[0, 1], ax=axes[0, 0])
axes[0, 0].set_title('Logistic Regression')
axes[0, 0].set_xlabel('Predicted')
axes[0, 0].set_ylabel('Actual')

# Random Forest
sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1], yticklabels=[0, 1], ax=axes[0, 1])
axes[0, 1].set_title('Random Forest')
axes[0, 1].set_xlabel('Predicted')
axes[0, 1].set_ylabel('Actual')

# Naive Bayes
sns.heatmap(cm_nb, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1], yticklabels=[0, 1], ax=axes[1, 0])
axes[1, 0].set_title('Naive Bayes')
axes[1, 0].set_xlabel('Predicted')
axes[1, 0].set_ylabel('Actual')

# Decision Tree
sns.heatmap(cm_dt, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1], yticklabels=[0, 1], ax=axes[1, 1])
axes[1, 1].set_title('Decision Tree')
axes[1, 1].set_xlabel('Predicted')
axes[1, 1].set_ylabel('Actual')

plt.tight_layout()
plt.show()
```


Logistic Regression
618
69
147
132
Random Forest
603
84
153
126
Naive Bayes
525
162
72
207
Decision Tree
554
133
146
133
Actual
Predicted

After reviewing the confusion matrix, I believe the inital selection model was in correct. We want to accurately predict turn for the customer base. This means we want a large amount of true positives and if given the choice we would want more false positives than false negatives. This is because offering deals to someone who wasn't going to leave initally will not make them leave, however getting a false negative will cause the customer to not be targeted with deals and could cause churn. Therefore, Naive Bayes is the best model for our problem.

In [ ]:

```

```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js