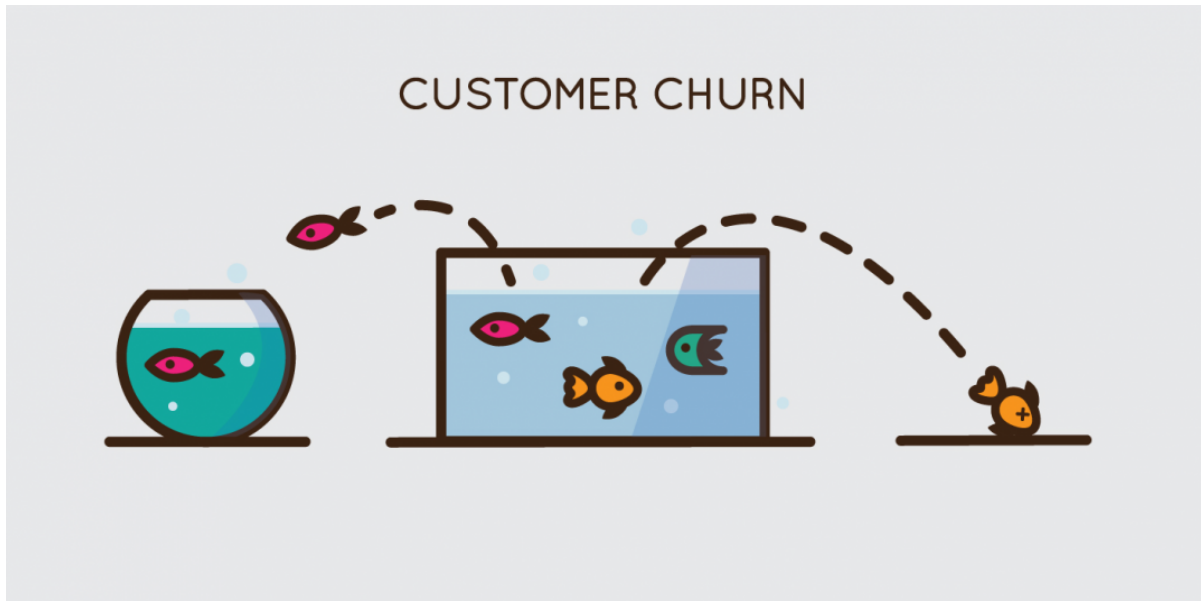# Phase 3 Project - Customer Churn Telco

- Name: Andrew Levinton
- Student Pace: Self Pace
- Instructor name: Ahbhineet Kukarni

I. Business Understanding

Business Problem:

- What does churn stand for? Churn in a business setting refers to losing an acquired, potentially profitable customer. The definition of churn can vary by industry (in Healthcare, dead people are considered churn while in finance, people with inactive cards are called churned).
- Why do businesses want to prevent churn? Acquiring a new customer is always more expensive than retaining an existing one. Hence, not letting them churn is the key to a sustained revenue stream.
- What metrics do we optimize on while predicting churn? F1-score and Recall are good ones, but you can also look at PR curves

The goals of this study will be:

- Analyze data from customer churn at Telcom - **metrics**
- Determine the best metrics to predict when a customer will/won't churn - **feature importance**
- Maximize Revenue per customer - **preventing churn**
- Make Reccomendations to Telcom to prevent churn - **Analyze the feature importance**

Business Questions to consider:

1. What percentage of customers leave after one month? 6 months? 1 year? 72 months(6 years, which is the max)?
2. Which services have the highest impact on customer churn?
3. Can we calculate lifetime value of a customer?
4. How do we prevent senior citizens from churning?
5. What types of customers buy into long term contracts? How does that impact churn?
6. What services do customers with longer tenure have?

Note This is the final notebook of the study. If you want to see a more detail analysis of the data and where the modeling and visualization come from, please visit the link to the github repository (https://github.com/andrewkoji/Phase_3_Project_churn.git) and look at the EDA notebook. You should be able to run both the final notebook and the EDA and get the same results.

Importing Necessary Libraries

In [95]:

```python
#data manipulation, imports
import pandas as pd
import numpy as np
from scipy import stats as stats
import math

#data visualization
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns
%matplotlib inline

# Classification algorithms
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
from xgboost import XGBClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler

# data visualiztion
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns

# machine learning libraries
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
precision_recall_fscore_support, f1_score, plot_confusion_matrix
from sklearn.model_selection import train_test_split, GridSearchCV,\
cross_validate, cross_val_predict, cross_val_score
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score

# pipeline libraries
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline as imbpipe
from sklearn.preprocessing import StandardScaler
from imblearn.pipeline import make_pipeline

import warnings # weird sns.distplot() warnings
warnings.filterwarnings("ignore")

plt.style.use('classic')
```

Functions used for study

In this project, some reports are reused time and time again, so we have defined functions to optimize the code of this project.

In [96]:

```python
# This function is used to visualize the feature importances of a machine l
# It takes a trained model as input and plots a horizontal bar chart
# the y-axis represents the features and the x-axis represents their corres
# The purpose of this function is to provide insights into which features a

def plot_feature_importances(model):
    n_features = X_processed_train.shape[1]
    plt.figure(figsize=(8,8))
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), X_processed_train.columns.values)
    plt.xlabel('Feature importance')
    plt.ylabel('Feature')


# This function generates a classification report that evaluates the perfor
# It takes the true labels (target_test) and predicted labels (target_pred)
# computes metrics such as precision, recall, F1-score, and support for eac
# The classification report provides a summary of the model's performance i

def classify(target_test, target_pred):
    # Generate the classification report
    report = classification_report(target_test, target_pred)

    # Print the classification report
    print("Classification Report:")
    print(report)

# This function plots the kernel density estimate (KDE) of a numerical feat
# It takes a DataFrame (df), the target variable (target), and the feature
# The function uses seaborn's kdeplot to visualize the distribution of the
# The purpose of this plot is to compare the distributions of a specific fe
# helping to identify potential differences or patterns that may be relevan

def kde_plot(df, target, feature):
    # Plotting distribution of tenure for churned and non-churned customers
    plt.figure(figsize=(10, 6))
    sns.kdeplot(df[df[target] == 0][feature], label='Non-Churned')
    sns.kdeplot(df[df[target] == 1][feature], label='Churned')
    plt.title(f'Distribution of {feature}')
    plt.xlabel(feature)
    plt.ylabel('Density')
    plt.xlim(0, df['tenure'].max())  # Adjusting x-axis limits
    plt.legend()
    plt.show()
```

II. Data Understanding

Next is the Data Understanding phase. Adding to the foundation of Business Understanding, it drives the focus to identify, collect, and analyze the data sets that can help you accomplish the project goals. This phase also has four tasks:

1. Collect initial data: Acquire the necessary data and (if necessary) load it into your analysis tool.
2. Describe data: Examine the data and document its surface properties like data format, number of records, or field identities.
3. Explore data: Dig deeper into the data. Query it, visualize it, and identify relationships among the data.
4. Verify data quality: How clean/dirty is the data? Document any quality issues.

1. Reading in Dataset

- Initial screening of the data will be checking each column in the dataset.
- Aspects to look for within the data will be datatypes. The datatypes will help determine if there is a need for say one hot encoding or any other type of data preparation.

Click to go to the dataset for more details on the column features, click here (https://www.kaggle.com/datasets/blastchar/telco-customer-churn)

```
In [97]:  #Read in dataset
          df = pd.read_csv('WA_Fn-UseC_-Telco-Customer-Churn.csv')
          #Checking first five rows
          df.head()
```

Out[97]:

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLi
0	7590-VHVEG	Female	0	Yes	No	1	No	No phe serv
1	5575-GNVDE	Male	0	No	No	34	Yes	
2	3668-QPYBK	Male	0	No	No	2	Yes	
3	7795-CFOCW	Male	0	No	No	45	No	No phe serv
4	9237-HQITU	Female	0	No	No	2	Yes	

5 rows × 21 columns

2. Explore data: Dig deeper into the data. Query it,

visualize it, and identify relationships among the data.

Defining target and feature variables

- In this section the dataframe will be split into its target and feature variables. The target will be defined as the 'churn' (whether the customer will leave or not).
- In the data exploration, general descriptive statistics like measures of central tendency (mean, median, mode) and variability (quartiles, standard deviation).

Telcom Customer Churn

- Each row represents a customer, each column contains customer's attributes described on the column Metadata.
- The raw data contains 7043 rows (customers) and 21 columns (features).

The "Churn" column is our target.

◀ ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ ▶

Dataframe of Unique Values

In [98]:

```python
output_data = []

for col in df.columns:

    # If the number of unique values in the column is less than or equal to
    if df.loc[:, col].nunique() <= 5:
        # Get the unique values in the column
        unique_values = df.loc[:, col].unique()
        # Append the column name, number of unique values, unique values, a
        output_data.append([col, df.loc[:, col].nunique(), unique_values, d
    else:
        # Otherwise, append only the column name, number of unique values,
        output_data.append([col, df.loc[:, col].nunique(),"-", df.loc[:, co

output_df = pd.DataFrame(output_data, columns=['Column Name', 'Number of Un

output_df
```

	Column Name	Number of Unique Values	Unique Values	Data Type
0	customerID	7043	-	object
1	gender	2	[Female, Male]	object
2	SeniorCitizen	2	[0, 1]	int64
3	Partner	2	[Yes, No]	object
4	Dependents	2	[No, Yes]	object
5	tenure	73	-	int64
6	PhoneService	2	[No, Yes]	object
7	MultipleLines	3	[No phone service, No, Yes]	object
8	InternetService	3	[DSL, Fiber optic, No]	object
9	OnlineSecurity	3	[No, Yes, No internet service]	object
10	OnlineBackup	3	[Yes, No, No internet service]	object
11	DeviceProtection	3	[No, Yes, No internet service]	object
12	TechSupport	3	[No, Yes, No internet service]	object
13	StreamingTV	3	[No, Yes, No internet service]	object
14	StreamingMovies	3	[No, Yes, No internet service]	object
15	Contract	3	[Month-to-month, One year, Two year]	object
16	PaperlessBilling	2	[Yes, No]	object
17	PaymentMethod	4	[Electronic check, Mailed check, Bank transfer...	object
18	MonthlyCharges	1585	-	float64
19	TotalCharges	6531	-	object
20	Churn	2	[No, Yes]	object

Checking descriptive statistics

```
In [99]:    #Checking Shape
            df.shape
```

Out[99]: (7043, 21)

In [100]: ▶ #checking descriptive statistics
df.describe()

Out[100]:

	SeniorCitizen	tenure	MonthlyCharges
count	7043.000000	7043.000000	7043.000000
mean	0.162147	32.371149	64.761692
std	0.368612	24.559481	30.090047
min	0.000000	0.000000	18.250000
25%	0.000000	9.000000	35.500000
50%	0.000000	29.000000	70.350000
75%	0.000000	55.000000	89.850000
max	1.000000	72.000000	118.750000

In [101]: ▶ # checking data types to see what methods of data cleaning or aggregation n
df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7043 entries, 0 to 7042
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   customerID        7043 non-null   object
 1   gender            7043 non-null   object
 2   SeniorCitizen     7043 non-null   int64
 3   Partner           7043 non-null   object
 4   Dependents        7043 non-null   object
 5   tenure            7043 non-null   int64
 6   PhoneService      7043 non-null   object
 7   MultipleLines     7043 non-null   object
 8   InternetService   7043 non-null   object
 9   OnlineSecurity    7043 non-null   object
 10  OnlineBackup      7043 non-null   object
 11  DeviceProtection  7043 non-null   object
 12  TechSupport       7043 non-null   object
 13  StreamingTV       7043 non-null   object
 14  StreamingMovies   7043 non-null   object
 15  Contract          7043 non-null   object
 16  PaperlessBilling  7043 non-null   object
 17  PaymentMethod     7043 non-null   object
 18  MonthlyCharges    7043 non-null   float64
 19  TotalCharges      7043 non-null   object
 20  Churn             7043 non-null   object
dtypes: float64(1), int64(2), object(18)
memory usage: 1.1+ MB
```

```
In [102]:  ▶| # checking for missing values in the dataframe to see if null removal or im
             df.isnull().sum()
```

Out[102]:
```
customerID          0
gender              0
SeniorCitizen       0
Partner             0
Dependents          0
tenure              0
PhoneService        0
MultipleLines       0
InternetService     0
OnlineSecurity      0
OnlineBackup        0
DeviceProtection    0
TechSupport         0
StreamingTV         0
StreamingMovies     0
Contract            0
PaperlessBilling    0
PaymentMethod       0
MonthlyCharges      0
TotalCharges        0
Churn               0
dtype: int64
```

In the dtypes above, we see that the total charges is listed as an object. For this we also see that there are some values that are listed as `''` which won't be able to convert to a float. For this we will change any value listed as `''` to a 0.0, then filter by all total charges above 0.0.

```
In [103]:  ▶| # changing Total Charges to a float
             df['TotalCharges'] = df['TotalCharges'].str.strip().apply(lambda x: 0.0 if
```

```
In [104]:  ▶| # filtering by Total charges above 0. Reasoning is the monthly charges are
             # total charges cannot be 0.
             df = df[df['TotalCharges'] > 0]
             df.describe()
```

Out[104]:

	SeniorCitizen	tenure	MonthlyCharges	TotalCharges
count	7032.000000	7032.000000	7032.000000	7032.000000
mean	0.162400	32.421786	64.798208	2283.300441
std	0.368844	24.545260	30.085974	2266.771362
min	0.000000	1.000000	18.250000	18.800000
25%	0.000000	9.000000	35.587500	401.450000
50%	0.000000	29.000000	70.350000	1397.475000
75%	0.000000	55.000000	89.862500	3794.737500
max	1.000000	72.000000	118.750000	8684.800000

```
In [105]:  ▶| #checking shape after filter
              df.shape
```

Out[105]: (7032, 21)

```
In [106]:  ▶| #lets check is there any duplicate records or not.
              df['customerID'].duplicated().sum()
```

Out[106]: 0

After filtering the dataset, it appears we only lost 9 customers which won't have a huge affect on the overall results.

Initial observarions:

- 7032 rows, 21 columns
- This means we have: 7032 customers, 20 feature variables, 1 target (churn)
- The max tenure (number of months that a customer has stayed with the company) is 72 months, with an average of 32 months.
- It would be interesting to see the types of contracts that people are on and see how it correlates with tenure.

```
In [107]:  ▶| df['Churn'].value_counts()
```

Out[107]: No   5163
           Yes   1869
           Name: Churn, dtype: int64

```
In [108]:  ▶ df['Churn'].hist()
             plt.title('Churned')

Out[108]:  Text(0.5, 1.0, 'Churned')
```
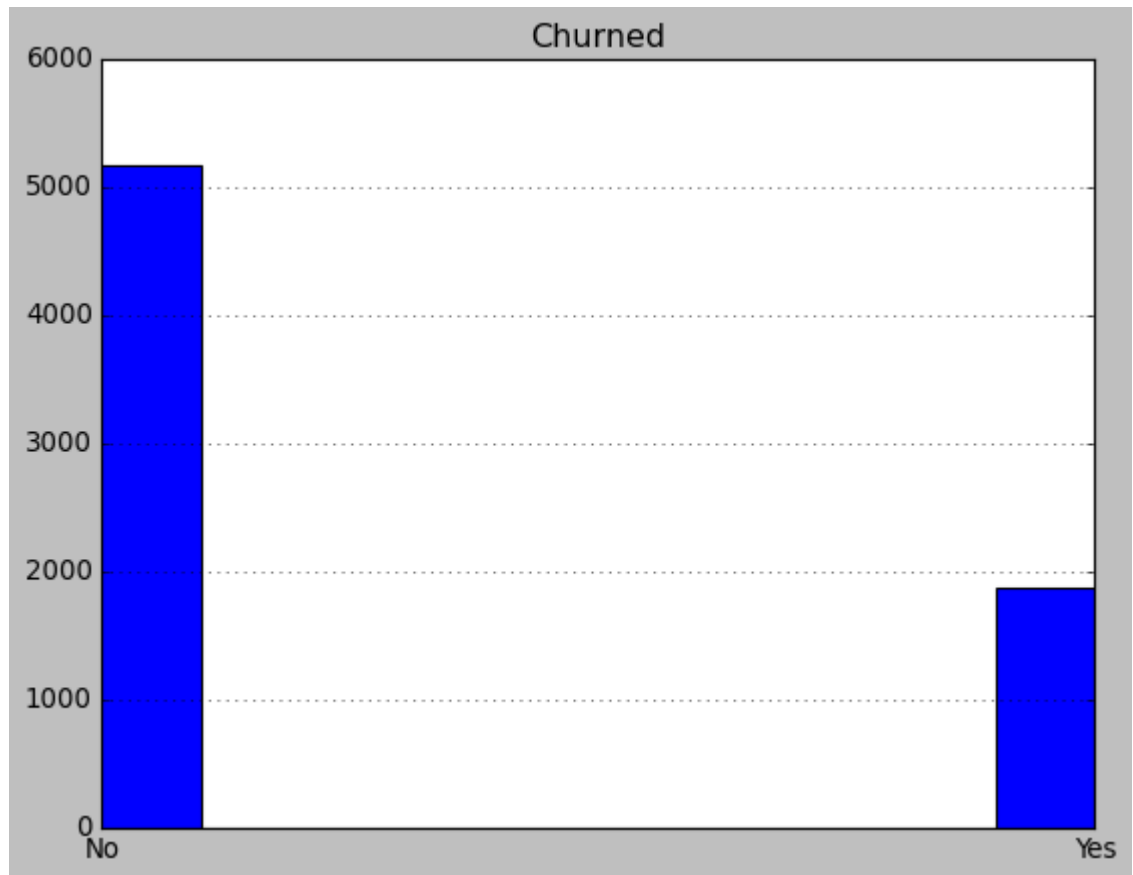

Data appears to be skewed towards "No" on the Churn, which shows that the majority of customers stay with the company in this dataset. Because the data appears to be skewed towards customers who did not churn, under or oversampling may be necessary here.

III. Data Preparation

A common rule of thumb is that 80% of the project is data preparation.

This phase, which is often referred to as "data munging", prepares the final data set(s) for modeling. It has five tasks:

- Select data: Determine which data sets will be used and document reasons for inclusion/exclusion.
- Clean data: Often this is the lengthiest task. Without it, you'll likely fall victim to garbage-in, garbage-out. A common practice during this task is to correct, impute, or remove erroneous values.
- Construct data: Derive new attributes that will be helpful. For example, derive someone's body mass index from height and weight fields.
- Integrate data: Create new data sets by combining data from multiple sources.

In [109]: ▶| `df.head()`

Out[109]:

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLir
0	7590-VHVEG	Female	0	Yes	No	1	No	No phe serv
1	5575-GNVDE	Male	0	No	No	34	Yes	
2	3668-QPYBK	Male	0	No	No	2	Yes	
3	7795-CFOCW	Male	0	No	No	45	No	No phe serv
4	9237-HQITU	Female	0	No	No	2	Yes	

5 rows × 21 columns

◀ ▬▬▬▬▬▬ ▶

Data Prep Step 1: Observe categorical data.

- In this section, we will observe the categorical data and replace it with numerical values.
- Examples that we see that are in common from the cell below are `"Yes"` and `"No"` from the columns for various services.
- Here we will replace `"Yes"` and `"No"` with `0` and `1` to enable feeding the data into a classifier.
- Remaining data will be dealt with likely OneHotEncoding

```
In [110]:  ▶  #checking the unique values of all column
               for col in df.columns:
                   if (df[col].dtypes=='object'):
                       print (f"{col}:{df[col].unique()}")
```

customerID:['7590-VHVEG' '5575-GNVDE' '3668-QPYBK' ... '4801-JZAZL' '8361-
LTMKD'
 '3186-AJIEK']
gender:['Female' 'Male']
Partner:['Yes' 'No']
Dependents:['No' 'Yes']
PhoneService:['No' 'Yes']
MultipleLines:['No phone service' 'No' 'Yes']
InternetService:['DSL' 'Fiber optic' 'No']
OnlineSecurity:['No' 'Yes' 'No internet service']
OnlineBackup:['Yes' 'No' 'No internet service']
DeviceProtection:['No' 'Yes' 'No internet service']
TechSupport:['No' 'Yes' 'No internet service']
StreamingTV:['No' 'Yes' 'No internet service']
StreamingMovies:['No' 'Yes' 'No internet service']
Contract:['Month-to-month' 'One year' 'Two year']
PaperlessBilling:['Yes' 'No']
PaymentMethod:['Electronic check' 'Mailed check' 'Bank transfer (automati
c)'
 'Credit card (automatic)']
Churn:['No' 'Yes']

Yes/No values will be simplified for the columns that have "No internet service" and "No" as
values. If the customer does not have internet service, then they will not have any internet
services...

- Here we will replace "No internet service" and "No phone service" with "No" .

```
In [111]:  ▶  df.replace({'No internet service':'No'},inplace=True)
               df.replace({'No phone service':'No'},inplace=True)
```

- Rechecking the unique values of all column after replacement values

```
In [112]:  ▶|  #checking the unique values of all column after replacement values
              for col in df.columns:
                  if (df[col].dtypes=='object'):
                      print (f"{col}:{df[col].unique()}")
```

```
customerID:['7590-VHVEG' '5575-GNVDE' '3668-QPYBK' ... '4801-JZAZL' '8361-
LTMKD'
 '3186-AJIEK']
gender:['Female' 'Male']
Partner:['Yes' 'No']
Dependents:['No' 'Yes']
PhoneService:['No' 'Yes']
MultipleLines:['No' 'Yes']
InternetService:['DSL' 'Fiber optic' 'No']
OnlineSecurity:['No' 'Yes']
OnlineBackup:['Yes' 'No']
DeviceProtection:['No' 'Yes']
TechSupport:['No' 'Yes']
StreamingTV:['No' 'Yes']
StreamingMovies:['No' 'Yes']
Contract:['Month-to-month' 'One year' 'Two year']
PaperlessBilling:['Yes' 'No']
PaymentMethod:['Electronic check' 'Mailed check' 'Bank transfer (automati
c)'
 'Credit card (automatic)']
Churn:['No' 'Yes']
```

Now that the data has been formatted, the classifier will need numerical data to assign. Here, we will do a simple replacement for all the "Yes", "No" data and assign them to 1 and 0.

```
In [113]:  ▶|  df.replace({'Yes':1},inplace=True)
              df.replace({'No':0},inplace=True)
```

```
In [114]:  ▶|  #checking the unique values of all column after replacement values
              for col in df.columns:
                  if (df[col].dtypes=='object'):
                      print (f"{col}:{df[col].unique()}")
```

```
customerID:['7590-VHVEG' '5575-GNVDE' '3668-QPYBK' ... '4801-JZAZL' '8361-
LTMKD'
 '3186-AJIEK']
gender:['Female' 'Male']
InternetService:['DSL' 'Fiber optic' 0]
Contract:['Month-to-month' 'One year' 'Two year']
PaymentMethod:['Electronic check' 'Mailed check' 'Bank transfer (automati
c)'
 'Credit card (automatic)']
```

Dropping the customerID column

The customer ID column will not serve any purpose in the data classification process, so it will be dropped for modeling purposes. If Telco wants to see the predictions of these customers,

In [115]: ▶| `df = df.drop('customerID', axis=1)`

Further exploration into the string predictors is needed to see if one hot encoding or string replacement of weighted values is necessary.

We will begin by separating the dataframe into dtypes and observing these features.

The tenure variable appears to be one of the more interesting continuous metrics in this data

In [116]: ▶| `df[['tenure', 'TotalCharges','MonthlyCharges']].sort_values('tenure', ascen`

Out[116]:

	tenure	TotalCharges	MonthlyCharges
2988	72	7880.25	109.65
3823	72	5763.15	78.85
3886	72	7677.40	106.85
6659	72	6719.90	92.30
6661	72	3784.00	53.65
...	...	...	...
3852	1	19.65	19.65
474	1	74.70	74.70
1371	1	79.20	79.20
1373	1	19.85	19.85
0	1	29.85	29.85

7032 rows × 3 columns

In [117]: ▶| `df.head()`

Out[117]:

	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetS
0	Female	0	1	0	1	0	0	
1	Male	0	0	0	34	1	0	
2	Male	0	0	0	2	1	0	
3	Male	0	0	0	45	0	0	
4	Female	0	0	0	2	1	0	Fib

Splitting data into target and predictors

- Identify the target variable: In this case, the target variable is the column named 'Churn'. The target variable represents the outcome or the variable we want to predict using the other columns in the dataset.
- Separate the predictors: The remaining columns in the dataset, excluding the target variable ('Churn'), are the predictors or independent variables. These columns contain the information that will be used to make predictions about the target variable.

To begin, the data needs to be split into `y = "Churn"` and `X`, which is the dataframe with the target variable dropped.

Train Test Split

- In machine learning, the train-test split is a technique used to evaluate the performance of a machine learning model. It involves `dividing the available dataset into two separate sets: the training set and the testing set.`
- The `training set` is used to train the model, meaning that the model learns patterns and relationships between the input features (independent variables) and the target variable (dependent variable). `The model tries to capture these patterns and generalize them to make predictions on unseen data.`

- The `testing set`, on the other hand, is used to evaluate the performance of the trained model. It serves as a proxy for new, unseen data that the model will encounter in the real world. By making predictions on the testing set, `the model's performance can be assessed by comparing its predicted outputs to the true target values in the testing set.`
- `The purpose of the train-test split is to assess how well the trained model generalizes to new, unseen data.` If the model performs well on the testing set, it suggests that the model has `learned meaningful patterns and is capable of making accurate predictions on new data.` However, `if the model performs poorly on the testing set, it indicates that the model may have overfit the training data, meaning that it has memorized the training examples instead of learning the underlying patterns.` In such cases, the model's ability to make accurate predictions on unseen data would likely be compromised.
  - In the context of the provided column names, the train-test split would involve `dividing the dataset into two subsets: one containing the columns related to the input features (e.g., gender, Internet service, contract type, payment method, etc.), and the other containing the target column 'Churn'`, which represents whether a customer has churned or not. The input features would be used to train the model, while the target column 'Churn' would be used to evaluate the model's performance. The split ensures that the model is assessed on its ability to predict churn accurately using the given features, enabling further analysis and improvements to the model if necessary.

```
In [118]:  ▶| X = df.drop('Churn', axis=1)
              target = df['Churn']
```

```
In [119]:  ▶| df.dtypes.unique()
```

```
Out[119]: array([dtype('O'), dtype('int64'), dtype('float64')], dtype=object)
```

```
In [120]:  ▶| X_train, X_test, target_train, target_test = train_test_split(X, target,
                                                                      test_si
```

Extracting numerical and categorical predictors

In this step of the data preparation process, we will separate the data into the categorical and numerical predictors. This will help to understand what needs to be done with the data before any model can be run on it.

```
In [121]:  ▶| #numerical predictors
              numerical_types = ['int64','float64']
              numerical_predictor_train = list(X_train.select_dtypes(include=numerical_ty
              numerical_predictor_test = list(X_test.select_dtypes(include=numerical_type
              #string(categorical) predictors
              object_types = ['O']
              object_predictor_train = list(X_train.select_dtypes(include=object_types))
              object_predictor_test = list(X_test.select_dtypes(include=object_types))
```

Looking at numerical predictors

```
In [122]:  ▶| X_train_numerical = X_train[numerical_predictor_train]
              X_test_numerical = X_test[numerical_predictor_test]
```

```
In [123]:  ▶| X_train_numerical.head()
```

Out[123]:

	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	OnlineSecurit
1980	0	0	0	10	1	1	
5485	0	0	0	1	1	0	
198	0	1	1	72	1	1	
6326	0	0	0	62	0	0	
1304	1	1	0	15	1	1	

In [124]: ▶| `X_test_numerical.head()`

Out[124]:

	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	OnlineSecurit
6830	0	0	0	35	1	0	
364	0	0	0	18	1	0	
2067	1	1	0	65	1	1	
6964	0	1	0	49	0	0	
4868	0	1	0	37	1	1	

The numerical data is currently "ready to go" for the machine learning models we intend to deploy. Next step is to deal with the string(categorical) predictors.

Extracting all string predictors

- Now that we have classified all of our predictors containing "Yes" and "No", we can now one hot encode the rest of the predictors that are categorical. These predictors are being one hot encoded because we want all the categories to hold the same weight in the model. Although, it may be worth exploring if under and over sampling needs to be done.
- The customer ID is dropped from the dataframe as it will not be used in the model as a predictor.

In [125]: ▶| `X_train_object = X_train[object_predictor_train]`
`X_test_object = X_test[object_predictor_test]`

In [126]: ▶| `X_train_object.head()`

Out[126]:

	gender	InternetService	Contract	PaymentMethod
1980	Male	Fiber optic	Month-to-month	Electronic check
5485	Male	DSL	Month-to-month	Mailed check
198	Male	Fiber optic	Two year	Bank transfer (automatic)
6326	Female	DSL	Two year	Credit card (automatic)
1304	Male	Fiber optic	Month-to-month	Bank transfer (automatic)

```
In [127]:    ▶| X_test_object.head()
```

Out[127]:

	gender	InternetService	Contract	PaymentMethod
6830	Male	0	Two year	Mailed check
364	Male	Fiber optic	Month-to-month	Bank transfer (automatic)
2067	Male	Fiber optic	One year	Bank transfer (automatic)
6964	Female	DSL	One year	Credit card (automatic)
4868	Female	Fiber optic	Month-to-month	Electronic check

Categorical data prep: One Hot Encoding

One hot encoding is a process used to represent categorical variables numerically, allowing them to be used as input in machine learning algorithms. The purpose of one hot encoding is to transform categorical data into a binary vector format that captures the distinct categories as binary values.

```
In [128]:    ▶| object_dummies_train = pd.get_dummies(X_train_object)
             object_dummies_test = pd.get_dummies(X_test_object)
```

Here is a quick check to make sure `df_numerical` and `object_dummies` have the same length before they are concatenated.

```
In [129]:    ▶| X_processed_train = pd.concat([object_dummies_train, X_train_numerical], ax
             X_processed_test = pd.concat([object_dummies_test, X_test_numerical], axis=
```

```
In [130]:    ▶| X_processed_train.head()
```

Out[130]:

	gender_Female	gender_Male	InternetService_0	InternetService_DSL	InternetService_Fiber optic
1980	0	1	0	0	
5485	0	1	0	1	
198	0	1	0	0	
6326	1	0	0	1	
1304	0	1	0	0	

5 rows × 27 columns

```
In [131]:  ▶ X_processed_test.head()
```

Out[131]:

	gender_Female	gender_Male	InternetService_0	InternetService_DSL	InternetService_Fi op
6830	0	1	1	0	
364	0	1	0	0	
2067	0	1	0	0	
6964	1	0	0	1	
4868	1	0	0	0	

5 rows × 27 columns

Standard Scaler

The StandardScaler operates on each feature independently and transforms it so that it has a mean of 0 and a standard deviation of 1. This process is also known as z-score normalization.

```
In [132]:  ▶ scaler = StandardScaler()

            X_processed_train_scaled = scaler.fit_transform(X_processed_train)
            X_processed_test_scaled = scaler.transform(X_processed_test)
```

IV. Modeling

What is widely regarded as data science's most exciting work is also often the shortest phase of the project.

Here you'll likely build and assess various models based on several different modeling techniques. This phase has four tasks:

Select modeling techniques: Determine which algorithms to try (e.g. regression, neural net). Generate test design: Pending your modeling approach, you might need to split the data into training, test, and validation sets. Build model: As glamorous as this might sound, this might just be executing a few lines of code like "reg = LinearRegression().fit(X, y)". Assess model: Generally, multiple models are competing against each other, and the data scientist needs to interpret the model results based on domain knowledge, the pre-defined success criteria, and the test design.

Baseline Model: Decision Tree Classifier

The main advantages of decision trees are their simplicity and interpretability. Decision trees are intuitive and easy to understand, as they mimic human decision-making processes by using a tree-like structure of if-else conditions. This structure allows for clear visualization and explanation of how the model reaches its predictions.

Here are the key purposes and benefits of using a decision tree classifier:

1. `Decision making and classification:` Decision trees provide a systematic approach to decision-making by considering a sequence of questions or conditions. At each internal node of the tree, a feature is evaluated, and the model chooses a path based on the feature's value. This process is repeated until a leaf node is reached, which corresponds to a predicted class or category.

2. `Feature importance:` Decision trees allow you to assess the importance of different features in the classification task. By examining the structure of the tree and the order of the features used, you can understand which features have the most significant impact on the classification outcome. This information can be valuable for feature selection and understanding the underlying data.

3. `Handling both numerical and categorical features:` Decision trees can handle a mixture of numerical and categorical features without requiring additional preprocessing, such as one-hot encoding. The algorithm automatically determines the best split points and conditions based on the feature types, making it convenient for datasets with diverse feature types.

4. `Nonlinear relationships:` Decision trees can capture nonlinear relationships between the features and the target variable. By recursively splitting the data based on different features and values, decision trees can create complex decision boundaries that can adapt to the data's intricacies.

5. `Handling missing values and outliers:` Decision trees can handle missing values and outliers by automatically determining the best splits based on available information. This eliminates the need for explicit imputation techniques or outlier removal, making the algorithm robust to missing or abnormal data points.

6. `Ensemble methods:` Decision trees can be combined into ensemble methods, such as random forests or gradient boosting, to improve predictive performance and reduce overfitting. These ensemble methods leverage the strengths of multiple decision trees to make more accurate predictions and improve generalization.

Build the tree

Here we will begin by building a model tree to eventually use in our random forest classifier. The goal will be to see how well the tree performs on our data to see if there is over or underfitting happening, then we will look to see what we can do with our random forest to improve the model.

In [133]: ▶
```
# Instantiate and fit a DecisionTreeClassifier
tree_clf = DecisionTreeClassifier(criterion='gini', max_depth=4, random_sta
tree_clf.fit(X_processed_train_scaled, target_train)
```

Out[133]: DecisionTreeClassifier(max_depth=4, random_state=42)

```
In [134]:  ▶| tree_clf.feature_importances_
```

Out[134]: array([0.00000000e+00, 0.00000000e+00, 3.71056844e-02, 0.00000000e+00,
       1.54300454e-01, 5.40400867e-01, 0.00000000e+00, 4.10975982e-03,
       0.00000000e+00, 0.00000000e+00, 4.81649787e-04, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 1.92291129e-01,
       8.28557705e-03, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 3.95337184e-03,
       0.00000000e+00, 2.36630915e-02, 3.54084152e-02])

```
In [135]:  ▶| plot_feature_importances(tree_clf)
```
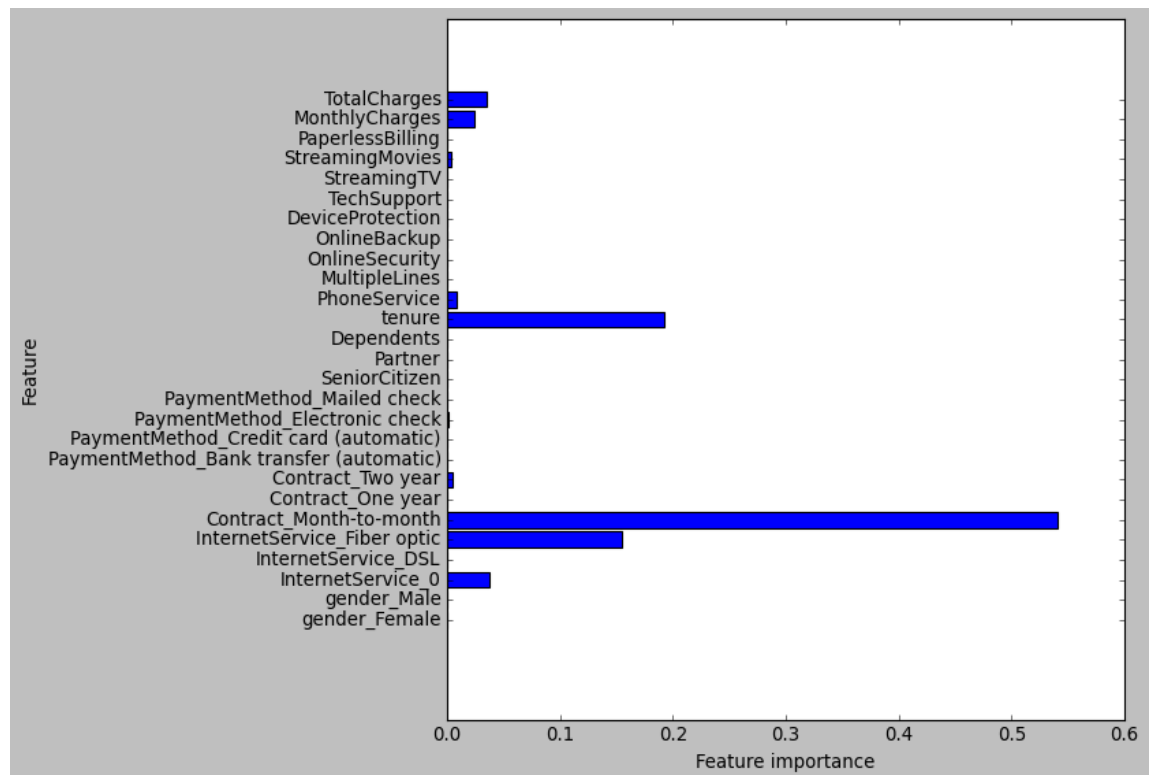

Observations:

- Contract Month-to-Month appears to have the highest effect on churn based on this tree
- Tenure(number of months in contract) appears to be the next most important feature for churn.

Test the tree

Here we will see how well the tree performs on the test set. We will grab a set of metrics to assess performance here. Metrics will include:

Confusion Matrix:

A **confusion matrix** is typically represented as a square matrix, where the rows correspond to the true classes and the columns correspond to the predicted classes. The diagonal elements of the matrix represent the correct predictions, while the off-diagonal elements represent the

incorrect predictions.

- Predicted Class
- | Positive | Negative |
- `True Class  |  True Positive  |  False Negative |`
- `False Class |  False Positive |  True Negative  |`

Confusion matrix interpretation: Defining True positive, True Negative, False Positive, False Negative

- True Positive (TP): Model predicts churned (1) and the actual label is churned (1).
- True Negative (TN): Model predicts non-churned (0) and the actual label is non-churned (0).
- False Positive (FP): Model predicts churned (1), but the actual label is non-churned (0).
- False Negative (FN): Model predicts non-churned (0), but the actual label is churned (1).

The case for observing False Positives or False Negatives

In the context of customer churn prediction, the severity of a false positive or a false negative can depend on the specific business or application scenario. However, generally speaking, false negatives tend to be considered more problematic in this case. Here's why:

- `False Positive (FP):` A **false positive** occurs when the model predicts that a customer will churn (1), but in reality, the customer does not churn (0). While false positives can lead to some inconveniences, such as targeting non-churned customers with retention efforts, they are usually less severe because the customer remains active and engaged. It may result in some extra costs or efforts, but it's preferable to err on the side of caution and take proactive measures.
- `False Negative (FN):` A **false negative** happens when the model predicts that a customer will not churn (0), but in reality, the customer does churn (1). False negatives are typically more concerning as they represent missed opportunities to take timely action and retain customers who are likely to churn. Losing a customer can result in revenue loss, negative impact on customer satisfaction, and potential cascading effects if they influence others to churn as well. Therefore, false negatives can have a more significant impact on the business's bottom line.

In customer churn prediction, the aim is going to be to `minimize false negatives`, which means identifying and retaining customers who are likely to churn. This is typically prioritized over false positives, where the extra effort spent on retaining customers who would not have churned may be seen as an acceptable trade-off.

Confusion matrix evaluation for False Negatives - Recall

- **Recall**, also known as sensitivity or true positive rate, measures the proportion of actual positive cases correctly identified by the model. It focuses on minimizing false negatives, which means it aims to avoid classifying positive cases as negative.
- By emphasizing recall in the context of customer churn analysis, the decision tree model aims to correctly identify as many churned customers as possible, minimizing the chances of missing important cases. This enables the business to take appropriate actions, such as offering incentives or personalized offers, to retain those customers and reduce churn rates.

In [136]:

```python
# Creating Test set predictions
predictions = tree_clf.predict(X_processed_test_scaled)

#plotting confusion matrix
cm = confusion_matrix(target_test, predictions, labels=tree_clf.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=tree_clf.classes_)
disp.plot()
plt.show()
```
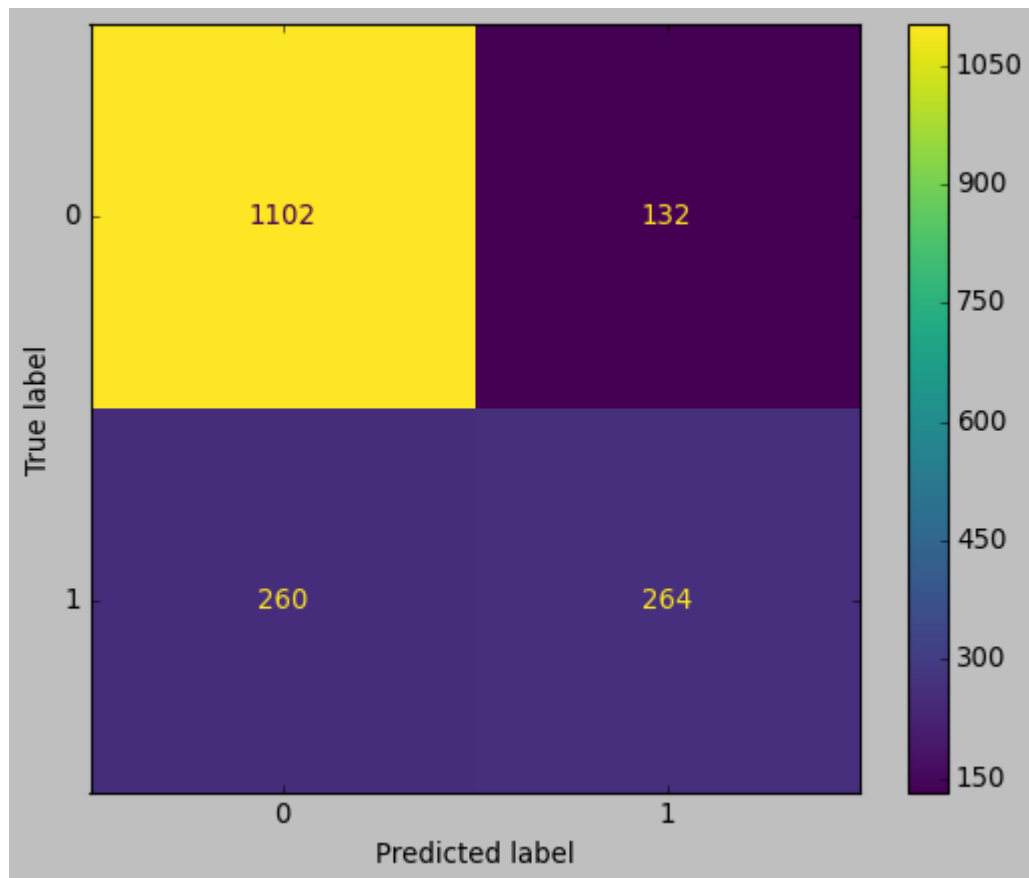
Metrics to be pulled from Confusion Matrix:

`Recall:` (also known as Sensitivity or True Positive Rate): It calculates the proportion of correctly predicted positive instances out of all actual positive instances. Recall is calculated as TP / (TP + FN).

To get these metrics, we will use the classification report function.

In [137]: ▶ `print(classification_report(target_test, predictions))`

```
              precision    recall  f1-score   support

           0       0.81      0.89      0.85      1234
           1       0.67      0.50      0.57       524

    accuracy                           0.78      1758
   macro avg       0.74      0.70      0.71      1758
weighted avg       0.77      0.78      0.77      1758
```

Interpreting the recall

The recall for class 0 (customers not churning) of 0.89 means that 89% of the customers who are not churning were correctly identified as such by the classifier. This indicates a relatively high level of accuracy in identifying customers who are likely to stay with the company.

The recall for class 1 (customers churning) of 0.50 means that only 50% of the customers who are actually churning were correctly identified as such by the classifier. This suggests that the classifier is less effective at identifying customers who are likely to churn, potentially resulting in a higher number of false negatives (customers who are churning but are not correctly classified as such).

Getting the testing accuracy

In [138]: ▶ `print("Testing Accuracy for Decision Tree Classifier: {:.4}%".format(accura`

```
Testing Accuracy for Decision Tree Classifier: 77.7%
```

78% accuracy on the test set means that the model correctly predicted the target variable for 78% of the instances in the test set. In other words, out of all the data points in the test set, approximately 78% were classified correctly by the decision tree.

Overfitting Detection

Decision trees have a tendency to **overfit** the training data, which means they may `memorize the training set too well and fail to generalize to new, unseen data`. Analyzing the ROC curves for the training and testing sets can help identify overfitting.

If the decision tree performs significantly better on the training set compared to the testing set (e.g., higher AUC-ROC for training), it could indicate overfitting. In such cases, you might need to consider regularization techniques like pruning or adjusting hyperparameters to reduce overfitting.

In [188]: ▶
```python
# Identify the optimal tree depth for given data
max_depths = list(range(1, 33))
train_results = []
test_results = []
for max_depth in max_depths:
    dt = DecisionTreeClassifier(criterion='entropy', max_depth=max_depth, r
    dt.fit(X_processed_train, target_train)
    train_pred = dt.predict(X_processed_train)
    false_positive_rate, true_positive_rate, thresholds = metrics.roc_curve
    roc_auc = metrics.auc(false_positive_rate, true_positive_rate)
    # Add auc score to previous train results
    train_results.append(roc_auc)
    target_pred = dt.predict(X_processed_test)
    false_positive_rate, true_positive_rate, thresholds = metrics.roc_curve
    roc_auc = metrics.auc(false_positive_rate, true_positive_rate)
    # Add auc score to previous test results
    test_results.append(roc_auc)

plt.figure(figsize=(12,6))
plt.plot(max_depths, train_results, 'b', label='Train AUC')
plt.plot(max_depths, test_results, 'r', label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('Tree depth')
plt.legend()
plt.show()
```
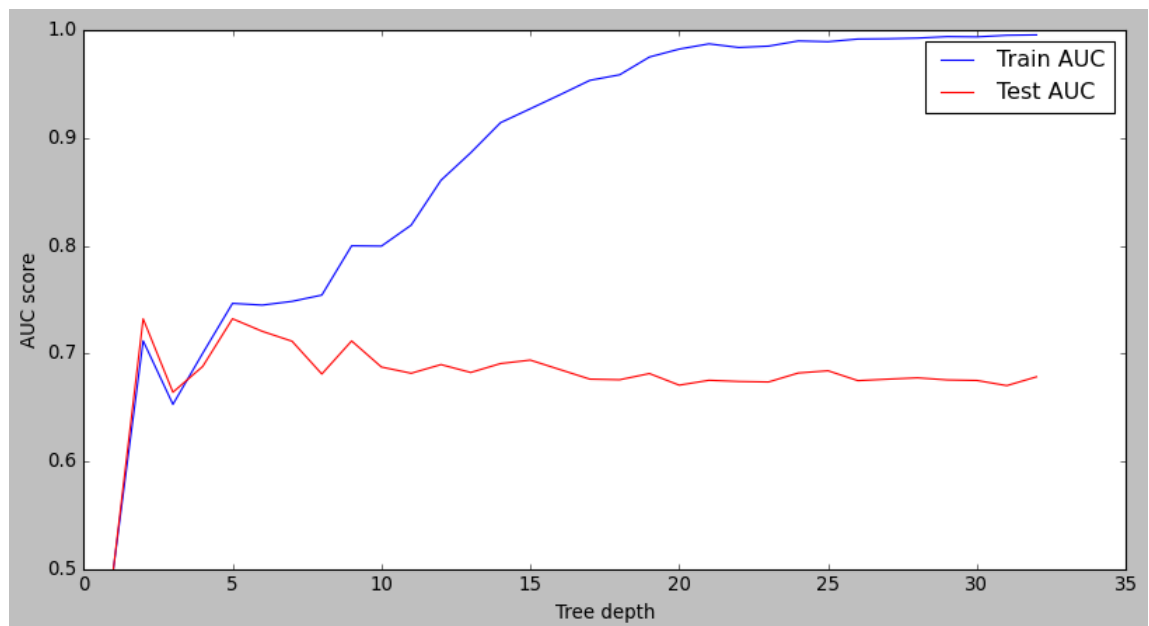

Based on the roc_auc curve, it appears that the decision tree becomes most susceptible to overfitting at an approximate tree depth higher than 5. We know this because the AUC shows a divergence between the training and testing data after a depth of 5.

Since we have a depth of 5 already in our classifier, we will not change that parameter.

Iteration 2: Bagged Trees

The first ensemble approach we'll try is a bag of trees. A bagging classifier, short for Bootstrap Aggregating classifier, is an ensemble learning method that combines multiple individual classifiers to make predictions. It is based on the idea of creating multiple subsets of the training data through bootstrapping, training a separate classifier on each subset, and then aggregating their predictions to obtain the final prediction.

Here we will instantiate a BaggingClassifier. First, initialize a DecisionTreeClassifier and set the same parameters that we did above for criterion and max_depth. Also set the n_estimators parameter for our BaggingClassifier to 20.

```python
In [141]: # Instantiate a BaggingClassifier
          bagged_tree = BaggingClassifier(DecisionTreeClassifier(criterion='gini', m
                                          n_estimators=20, random_state=42)
```

Scoring the Bagging Classifier

Here we will check how the bagged tree performs on the training and testing set.

```python
In [142]: # Fit to the training data
          bagged_tree.fit(X_processed_train_scaled, target_train)
```
```
Out[142]: BaggingClassifier(base_estimator=DecisionTreeClassifier(max_depth=5),
                            n_estimators=20, random_state=42)
```

```python
In [143]: # Training accuracy score
          bagged_tree.score(X_processed_train_scaled, target_train)
```
```
Out[143]: 0.8077360637087599
```

```python
In [144]: # Test accuracy score
          bagged_tree.score(X_processed_test_scaled, target_test)
```
```
Out[144]: 0.7878270762229806
```

In [145]:

```python
# Predict the labels for the test data
y_pred = bagged_tree.predict(X_processed_test_scaled)

# Obtain the confusion matrix
confusion_mat = confusion_matrix(target_test, target_pred)

# Display the confusion matrix
# Define class labels
class_labels = ["Not Churned", "Churned"]
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_mat, annot=True, cmap="Blues", fmt="d")
plt.title("Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.xticks([0.5, 1.5], class_labels)
plt.yticks([0.5, 1.5], class_labels)
plt.show()
```
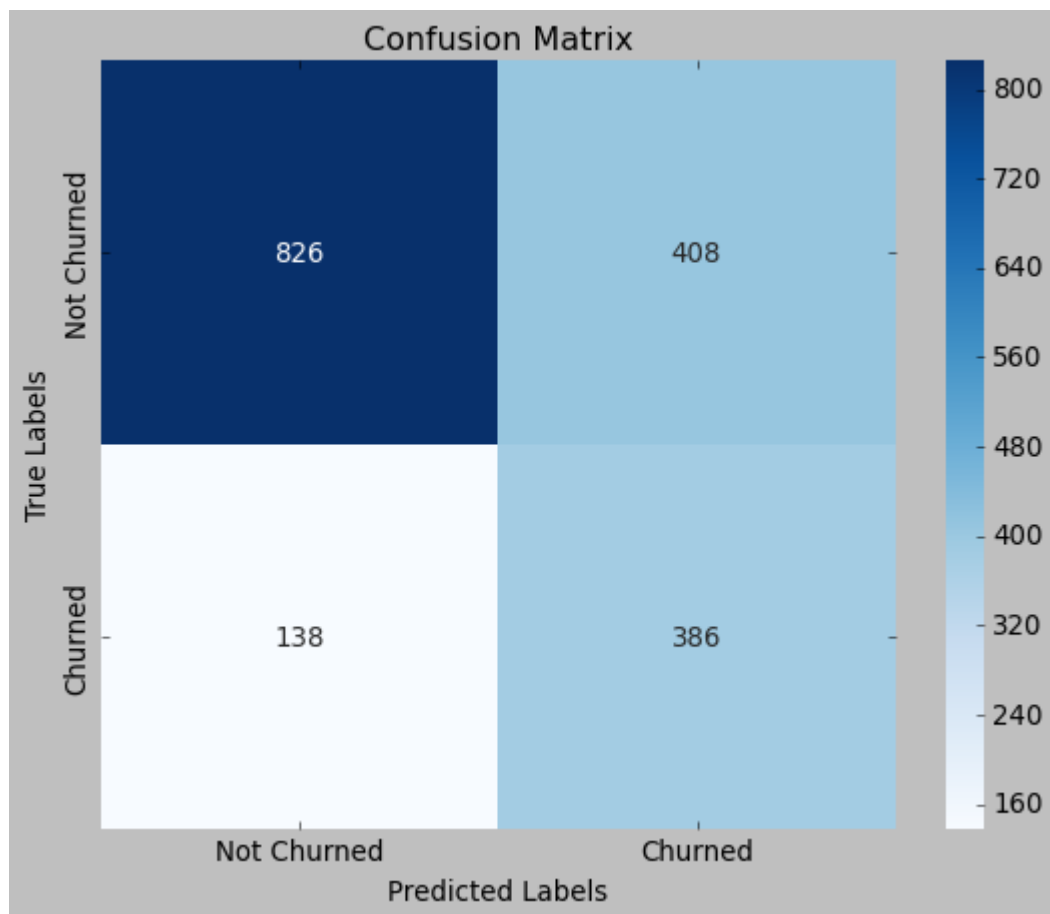
```
In [146]:   ▶ classify(target_test, target_pred)
```

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

           0       0.86      0.67      0.75      1234
           1       0.49      0.74      0.59       524

    accuracy                           0.69      1758
   macro avg       0.67      0.70      0.67      1758
weighted avg       0.75      0.69      0.70      1758
```

Analysis

- For class 0 (not churned), a recall of 0.92 means that 92% of the actual not churned customers were correctly identified as such by the classifier. This indicates a high level of accuracy in recognizing customers who are likely to stay with the company.
- For class 1 (churned), a recall of 0.49 means that only 49% of the actual churned customers were correctly identified as such by the classifier. This suggests that the classifier is less effective at capturing churned customers, potentially resulting in a higher number of false negatives (customers who are churned but are not correctly classified as such).

Iteration 3: Random Forest

A random forest is an ensemble learning method that combines multiple decision trees to make predictions. It is designed to improve the accuracy and robustness of predictions by introducing randomness and diversity in the learning process.

The purpose of a random forest is to address some of the limitations of individual decision trees, such as high variance and overfitting. It achieves this by leveraging the power of multiple trees and aggregating their predictions to obtain a final prediction. The random forest algorithm consists of the following key concepts:

- Ensemble of Decision Trees: A random forest consists of a collection or ensemble of decision trees. Each tree is trained independently on a random subset of the training data.
- Random Feature Subsets: In addition to using random subsets of the training data, a random forest also employs a technique called "feature bagging" or "feature subsampling." At each split in a decision tree, only a random subset of features is considered for determining the best split. This random feature selection enhances the diversity among the trees and reduces the correlation between them.
- Bootstrap Aggregation (Bagging): Each decision tree in the random forest is trained on a bootstrap sample of the training data. Bootstrap sampling involves randomly selecting instances from the training data with replacement. This process creates different subsets of the data for each tree, allowing them to have variations in the training data they observe.
- Voting or Averaging Predictions: When making predictions, each tree in the random forest independently predicts the target variable. For classification tasks, the class with the majority of votes is selected as the final prediction. For regression tasks, the predictions of

all trees are averaged to obtain the final prediction.

The advantages of random forests include:

- Improved Accuracy: Random forests typically offer higher accuracy compared to individual decision trees. By combining the predictions of multiple trees, they can effectively reduce bias and variance, leading to more accurate predictions.
- Robustness: Random forests are less prone to overfitting since the randomness in feature selection and training data subsets helps to reduce the model's sensitivity to noise and outliers.
- Variable Importance: Random forests provide a measure of variable importance. By analyzing the average decrease in impurity or information gain caused by each feature, you can identify the most important features in the dataset.

Here we will initialize a random forest classifier and score it on the training and testing set.

In [147]: ▶ ```
# Instantiate and fit a RandomForestClassifier
forest = RandomForestClassifier(n_estimators=10, max_depth= 5, random_state
forest.fit(X_processed_train_scaled, target_train)
```

Out[147]: ```
RandomForestClassifier(class_weight='balanced', max_depth=5, n_estimators=
10,
                       random_state=42)
```

In [148]: ▶ ```
# Test accuracy score
forest.score(X_processed_train_scaled, target_train)
```

Out[148]: 0.7383390216154722

In [149]: ▶ ```
# Test accuracy score
forest.score(X_processed_test_scaled, target_test)
```

Out[149]: 0.7497155858930603

Results here show a similar pattern to the decision tree classifier, further exploration on the forest's model evaluation will now be assessed to see if there is a difference in the feature importances.

In [150]: ▶| `plot_feature_importances(forest)`
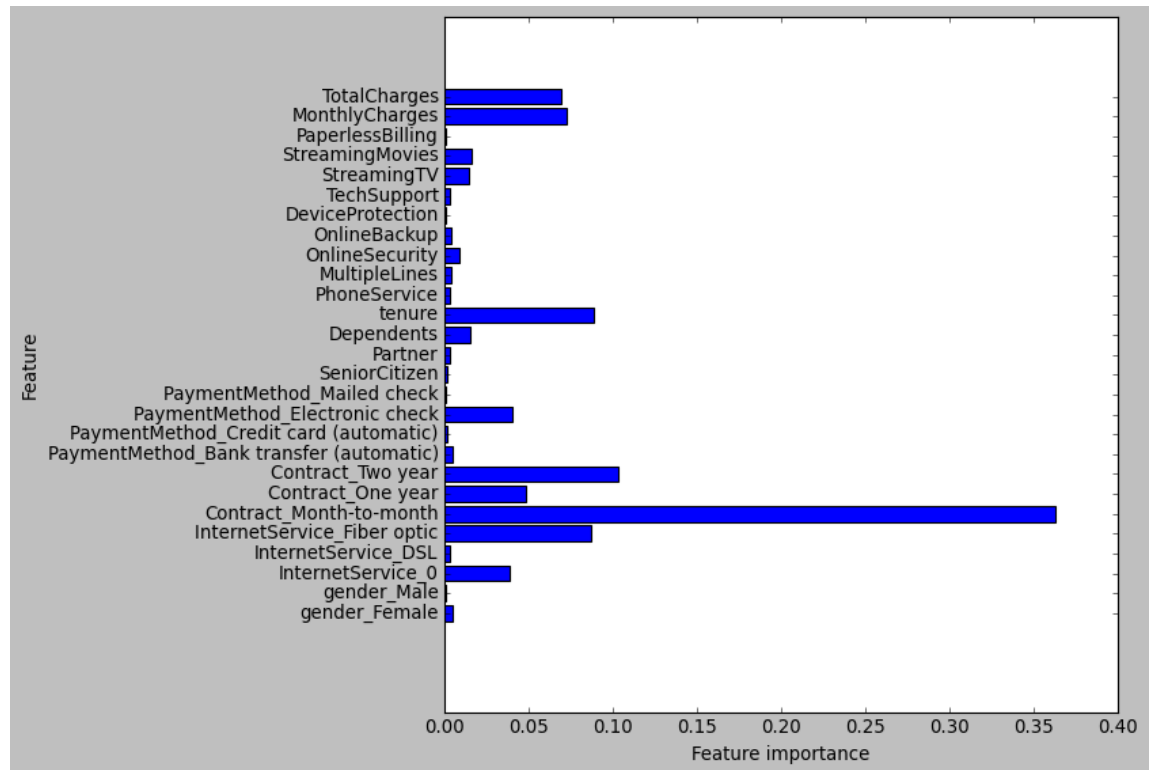
```
In [151]:  # Define class labels
           class_labels = ["Not Churned", "Churned"]

           # Obtain the confusion matrix
           confusion_mat = confusion_matrix(target_test, y_pred)

           # Display the confusion matrix
           plt.figure(figsize=(8, 6))
           sns.heatmap(confusion_mat, annot=True, cmap="Blues", fmt="d")
           plt.title("Random Forest Classifier")
           plt.xlabel("Predicted Labels")
           plt.ylabel("True Labels")
           plt.xticks([0.5, 1.5], class_labels)
           plt.yticks([0.5, 1.5], class_labels)
           plt.show()
```
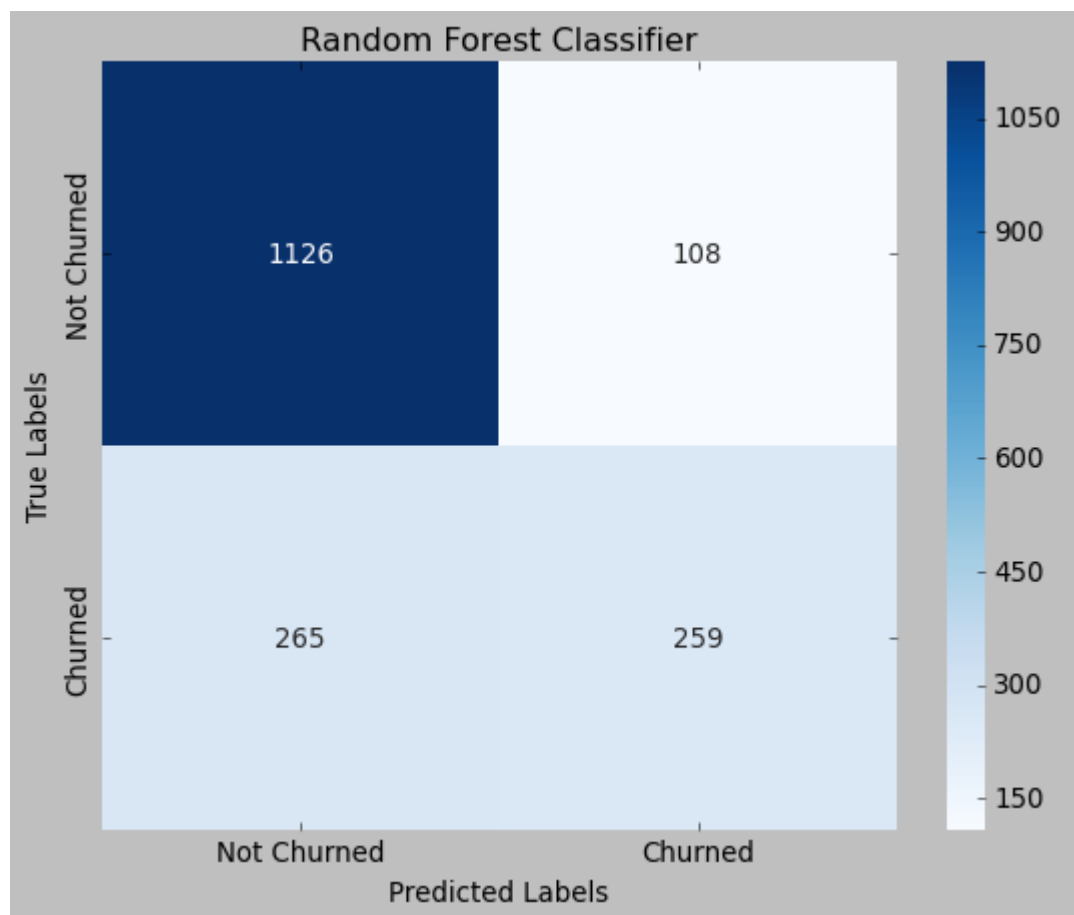
```python
In [152]:   # Generate the classification report
            report = classification_report(target_test, target_pred)

            # Print the classification report
            print("Classification Report:")
            print(report)
```

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

           0       0.86      0.67      0.75      1234
           1       0.49      0.74      0.59       524

    accuracy                           0.69      1758
   macro avg       0.67      0.70      0.67      1758
weighted avg       0.75      0.69      0.70      1758
```

Observations

In comparison to the decision tree, we see that the random forest classifier makes better use of various features in the dataset, whereas the decision tree relied more heavily on the `contract month-to-month` variable and `tenure` for its classifications. The next step will be to implement a **pipeline** to the model.

Iteration 4: XGBoost Pipeline

XGBoost (eXtreme Gradient Boosting) is a powerful and widely used machine learning algorithm designed for supervised learning tasks, including both classification and regression problems. It is an implementation of the gradient boosting framework that has gained popularity due to its exceptional performance and scalability.

The purpose of XGBoost is to create an accurate and robust predictive model by combining multiple weak predictive models, typically decision trees. Here are some key purposes and advantages of using XGBoost:

1. `High Prediction Accuracy:` XGBoost is known for its exceptional accuracy and predictive power. It leverages gradient boosting techniques to iteratively train a series of weak learners (decision trees) that sequentially correct the mistakes made by the previous models. This iterative process allows XGBoost to capture complex relationships within the data, resulting in improved predictive performance.
2. `Handling Complex Data Patterns:` XGBoost can effectively handle complex data patterns, including non-linear relationships and interactions between features. The algorithm can automatically capture and model these intricate relationships through the ensemble of decision trees, making it a suitable choice for a wide range of machine learning problems.
3. `Regularization and Control over Model Complexity:` XGBoost provides various regularization techniques to control the complexity of the model and prevent overfitting. Regularization methods, such as L1 and L2 regularization, can be applied to the model's weights or the structure of the decision trees. This helps prevent the model from becoming too complex and provides a way to balance between overfitting and underfitting.

4. `Feature Importance Analysis:` XGBoost offers built-in methods to assess the importance of features in the predictive model. By examining the contribution of each feature in the ensemble of decision trees, you can gain insights into the most influential features for making predictions. This analysis aids in feature selection, understanding the underlying data, and interpreting the model's behavior.

5. `Scalability and Efficiency:` XGBoost is designed to be highly scalable and efficient, enabling it to handle large datasets and perform computations in parallel. The algorithm includes optimizations such as parallel tree construction, approximate algorithms for split finding, and efficient memory usage. These features make XGBoost suitable for both small and large-scale machine learning tasks.

6. `Flexibility and Customization:` XGBoost offers a wide range of hyperparameters that can be tuned to optimize the model's performance for specific tasks. You can adjust parameters related to the tree structure, learning rate, regularization, and more. This flexibility allows you to customize the algorithm to suit your specific needs and achieve the best results.

In [161]:
```python
xgb_pipe = imbpipe(steps=[
    ('ss', StandardScaler()),
    ('sm', SMOTE(random_state=42)),
    ('xgb', XGBClassifier(random_state=42,
                          max_depth = 5,
                          tree_method='hist',
                          n_estimators = 50,
                          n_jobs = -1))
])

# Train the model
xgb_pipe.fit(X_processed_train, target_train)

# Make predictions on the test set
target_pred = xgb_pipe.predict(X_processed_test)

# Evaluate the model
accuracy = accuracy_score(target_test, target_pred)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

Accuracy: 79.69%

```
In [162]:  ▶| # # Define the XGBoost model
           # xgb_pipeline = xgb.XGBClassifier(
           #     n_estimators=10,  # Number of boosting iterations
           #     max_depth=5,  # Maximum depth of each tree
           #     learning_rate=0.1,  # Learning rate
           #     subsample=0.8,  # Subsample ratio of the training instances
           #     colsample_bytree=0.8,  # Subsample ratio of columns when constructing
           #     random_state=42
           # )

           # # Train the model
           # xgb_pipeline.fit(X_processed_train, target_train)

           # # Make predictions on the test set
           # target_pred = xgb_pipeline.predict(X_processed_test)

           # # Evaluate the model
           # accuracy = accuracy_score(target_test, target_pred)
           # print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

```python
In [164]:  ▶| def plot_feature_importances(model, feature_names):
               importance = model.get_booster().get_score(importance_type='weight')
               feature_importance = []

               for feature, score in importance.items():
                   feature_importance.append((feature, score))

               feature_importance.sort(key=lambda x: x[1], reverse=True)
               features, scores = zip(*feature_importance)

               # Retrieve actual column names
               actual_features = [feature_names[int(feature[1:])] for feature in featu

               plt.figure(figsize=(8, 8))
               plt.barh(range(len(actual_features)), scores, align='center')
               plt.yticks(range(len(actual_features)), actual_features)
               plt.xlabel('Feature Importance')
               plt.ylabel('Feature')
               plt.title('Feature Importances')
               plt.show()

           # Call the function with the trained model and column names
           column_names = X_processed_train.columns.tolist()
           plot_feature_importances(xgb_pipe.named_steps['xgb'], column_names)
```
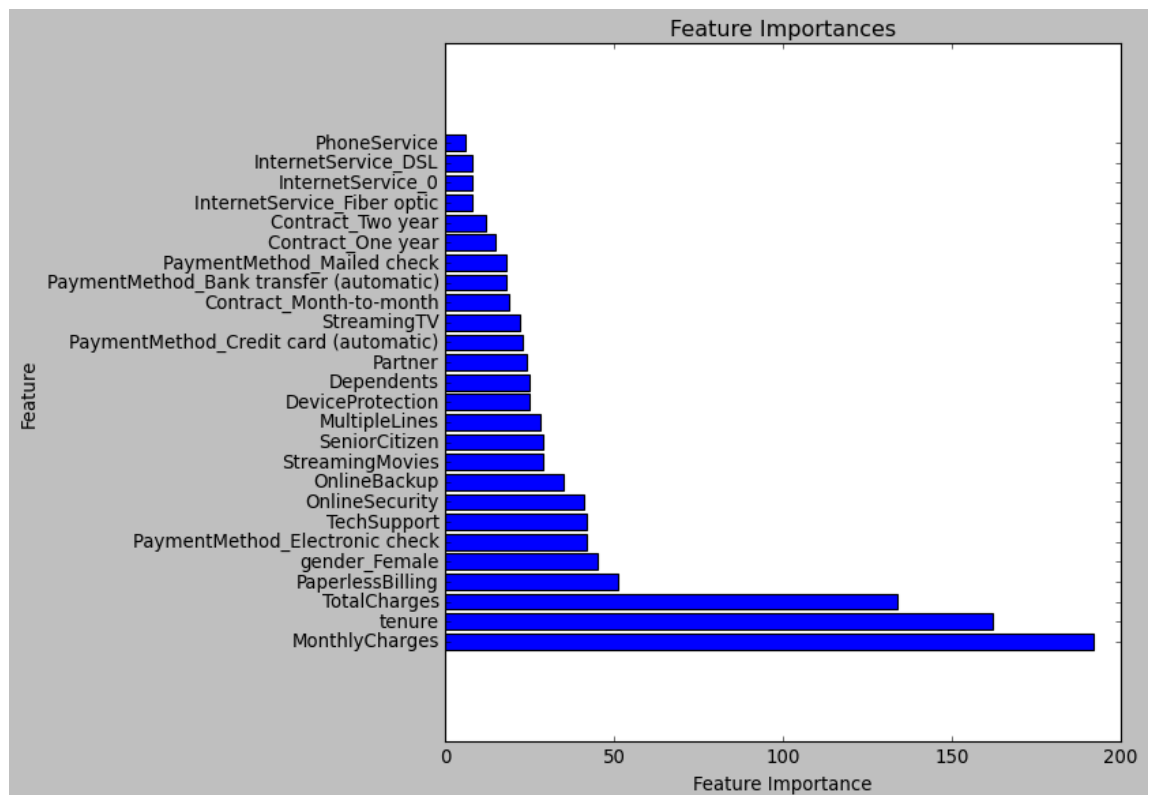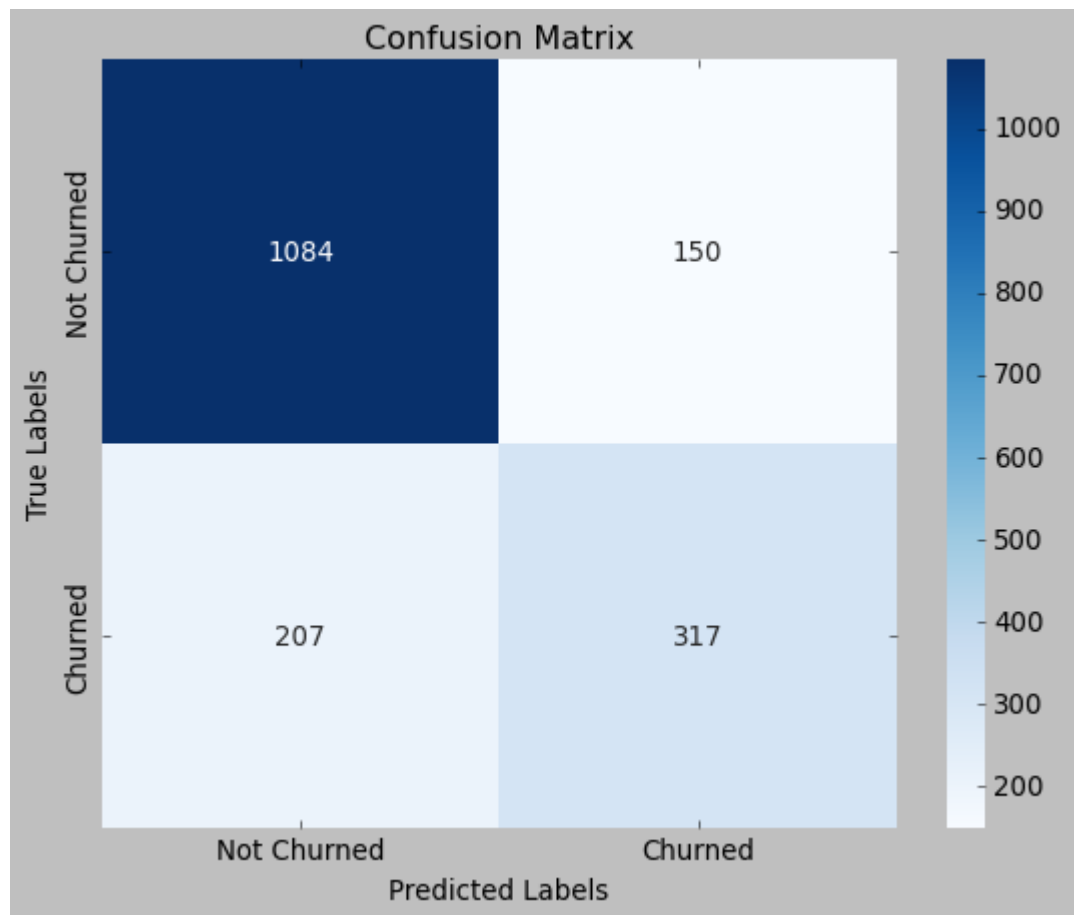
In [165]:

```python
# Define class labels
class_labels = ["Not Churned", "Churned"]

# Obtain the confusion matrix
confusion_mat = confusion_matrix(target_test, target_pred)

# Display the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_mat, annot=True, cmap="Blues", fmt="d")
plt.title("Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.xticks([0.5, 1.5], class_labels)
plt.yticks([0.5, 1.5], class_labels)
plt.show()
```


In [1519]:

```python
classify(target_test, target_pred)
```

Classification Report:

	precision	recall	f1-score	support
0	0.81	0.92	0.86	1234
1	0.72	0.49	0.58	524
accuracy			0.79	1758
macro avg	0.77	0.71	0.72	1758
weighted avg	0.78	0.79	0.78	1758

Iteration 4: Random Forest Pipelines

A pipeline refers to a sequence of data processing steps that are chained together to form a cohesive workflow. It is an important concept because it allows for a systematic and automated approach to data preprocessing, feature engineering, model training, and evaluation.

The key benefits of using a pipeline in machine learning are:

1. **Code modularity and reusability**: By encapsulating different stages of the machine learning workflow into separate components, pipelines enable code modularity and reusability. Each step can be implemented as a separate module, making it easier to maintain and modify specific parts of the pipeline without affecting the entire workflow.
2. **Reproducibility**: Pipelines ensure that the same sequence of steps is consistently applied to the data, allowing for reproducibility of the results. This is especially important when sharing and collaborating on machine learning projects, as it helps others reproduce the same analysis with ease.
3. **Automation and efficiency**: Pipelines automate the process of executing multiple sequential tasks, reducing manual effort and increasing efficiency. Once a pipeline is set up, it can be applied to new datasets or updated with minimal effort, saving time and resources.

First, we will create a pipeline to test on the dataset and see how it compares with the random forest.

In the dataset, we will use a `standard scaler` with our random forest classifier. The `StandardScaler` from sklearn.preprocessing is a commonly used class in scikit-learn for standardizing numerical features in a dataset. It applies a common scaling transformation to each feature, making them have zero mean and unit variance. This process is also known as z-score normalization or standardization.

Here are the key aspects and functionalities of the StandardScaler:

1. `Centering and scaling:` The StandardScaler standardizes features by subtracting the mean value of each feature and dividing it by the standard deviation. This ensures that the resulting features have zero mean and unit variance.
2. `Fit and transform methods:` The StandardScaler follows the scikit-learn API convention and provides the fit and transform methods. The fit method calculates the mean and standard deviation of each feature from the training data. The transform method applies the scaling transformation using the computed mean and standard deviation values.
3. `Fit-transform convenience method:` To streamline the preprocessing workflow, the StandardScaler also offers the fit_transform method, which combines the fitting and transformation steps into a single operation.
4. `Handling missing values:` The StandardScaler automatically handles missing values by excluding them during the fit step and replacing them with the mean value of the corresponding feature during the transform step.

```
In [166]:  ▶| # Instantiate the RandomForestClassifier
           forest = RandomForestClassifier(n_estimators=10, max_depth=5, random_state=

           # Instantiate the SMOTE object
           smote = SMOTE(random_state=42)

           # Define the pipeline
           forest_pipe = make_pipeline(
               StandardScaler(),
               smote,
               forest
           )

           # Fit the pipeline on the training data
           forest_pipe.fit(X_processed_train, target_train)

           # Make predictions on the test data
           target_pred = forest_pipe.predict(X_processed_test)
```
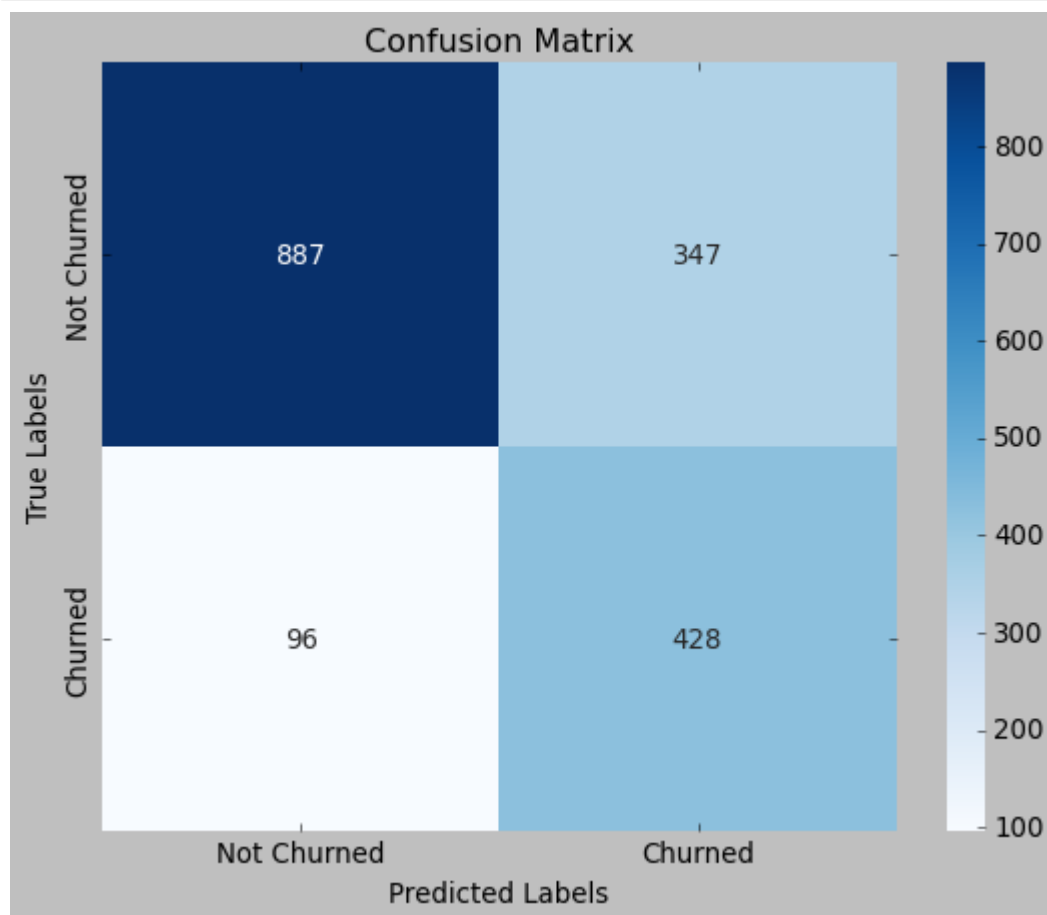
```
In [167]:  ▶| # Calculate the score on test data
           forest_pipe.score(X_processed_test, target_test)
```

Out[167]: 0.7480091012514221

```python
# Calculate the confusion matrix
confusion_mat = confusion_matrix(target_test, target_pred)

# Define class labels
class_labels = ["Not Churned", "Churned"]

# Display the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_mat, annot=True, cmap="Blues", fmt="d")
plt.title("Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.xticks([0.5, 1.5], class_labels)
plt.yticks([0.5, 1.5], class_labels)
plt.show()
```

```
classify(target_test, target_pred)
```

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

           0       0.90      0.72      0.80      1234
           1       0.55      0.82      0.66       524

    accuracy                           0.75      1758
   macro avg       0.73      0.77      0.73      1758
weighted avg       0.80      0.75      0.76      1758
```

```python
# Access the RandomForestClassifier model within the pipeline
forest_model = forest_pipe.named_steps['randomforestclassifier']

# Retrieve feature importances
importances = forest_model.feature_importances_

# Get the names of the features
feature_names = X_processed_train.columns

# Sort the feature importances in descending order
indices = np.argsort(importances)[::-1]

# Plot the feature importances as a horizontal bar chart
plt.figure(figsize=(10, 6))
plt.title("Feature Importances")
plt.barh(range(len(importances)), importances[indices], align="center")
plt.yticks(range(len(importances)), feature_names[indices])
plt.xlabel("Importance")
plt.ylabel("Features")
plt.tight_layout()
plt.show()
```
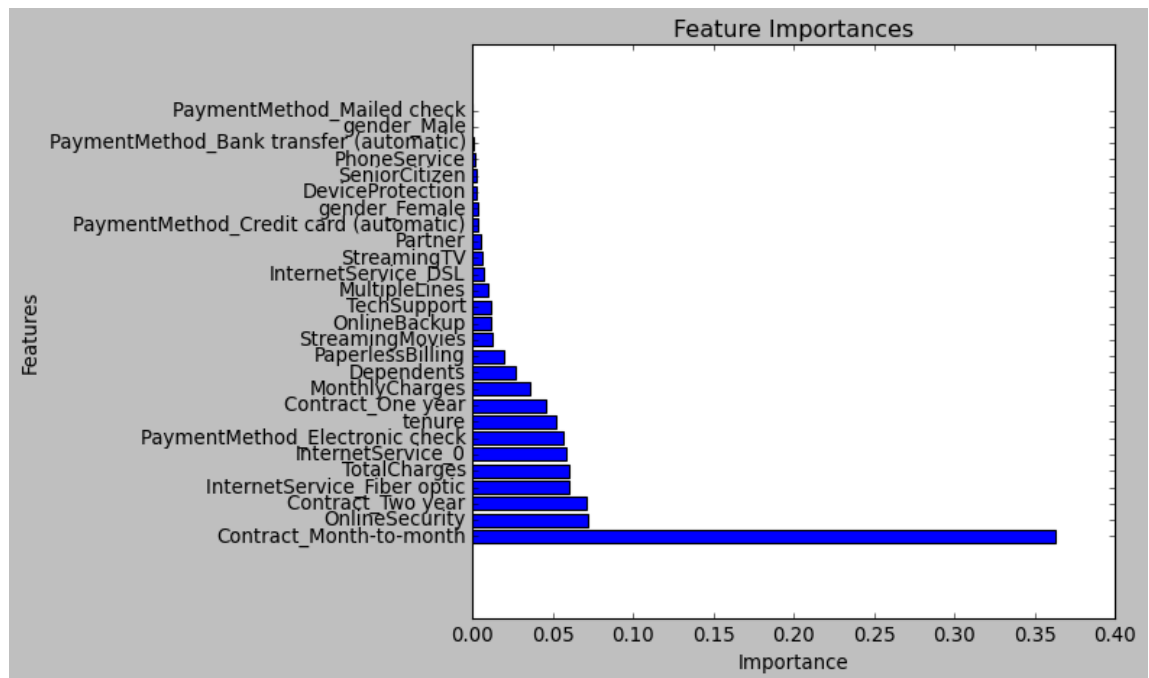

Iteration 5: Adding in Parameter Grid

The purpose of a parameter grid for a pipeline is to define a set of hyperparameter values to be tested during the process of hyperparameter tuning. When using a pipeline, which combines multiple steps in the machine learning workflow, including data preprocessing and model training, a parameter grid allows you to specify different values for the hyperparameters of each step in the pipeline.

The key purposes of a parameter grid for a pipeline are as follows:

1. `Hyperparameter tuning:` The primary purpose of a parameter grid is to enable hyperparameter tuning. By defining a range of values for each hyperparameter in the grid, the grid search algorithm systematically explores different combinations of hyperparameters to find the best set of hyperparameters that maximizes the model's performance.
2. `Automation and efficiency:` With a parameter grid, you can automate the process of trying out various hyperparameter combinations. Instead of manually iterating through all the possible combinations, the grid search algorithm handles this process for you, saving time and effort. This automation increases efficiency by systematically searching the hyperparameter space and evaluating the model's performance for each combination.
3. `Performance optimization:` The parameter grid allows you to search for the best hyperparameter values that optimize the model's performance on a specific metric or objective, such as accuracy, precision, or recall. By trying different hyperparameter values, you can fine-tune the model and improve its performance.
4. `Generalization and robustness:` By testing multiple hyperparameter combinations, the parameter grid helps in finding hyperparameters that generalize well to unseen data. It reduces the risk of overfitting by preventing the model from being too specialized to the training data, resulting in a more robust and reliable model.

Here are the parameters we will be tuning in the model:

*   `'forest__n_estimators'` : This parameter represents the number of decision trees in the random forest.
*   `'forest__max_depth'` : This parameter controls the maximum depth of each decision tree in the random forest. A larger max_depth allows the trees to grow deeper and capture more complex relationships in the data. Setting it to None means there is no maximum depth limit.
*   `'forest__min_samples_split'` : This parameter determines the minimum number of samples required to split an internal node in a decision tree. It controls the process of splitting a node based on the number of samples it contains.
*   `'forest__min_samples_leaf'` : This parameter specifies the minimum number of samples required to be at a leaf node (i.e., a terminal node) in a decision tree. It controls the creation of leaf nodes by setting a threshold on the minimum number of samples in a leaf.

Using GridSearchCV

*   `GridSearchCV` is a class in scikit-learn that performs an exhaustive search over a specified parameter grid to find the best hyperparameters for a given machine learning model. It is a technique for hyperparameter tuning, which involves finding the optimal combination of hyperparameters that maximizes the model's performance.
*   `Cross-validation:` GridSearchCV performs cross-validation, which is a technique for evaluating the model's performance on multiple subsets of the training data. It splits the training data into multiple folds, trains the model on a subset of the folds, and evaluates its performance on the remaining fold. This process is repeated for each fold, and the results are averaged to get an overall performance estimate.

```python
In [171]:   # Instantiate the RandomForestClassifier
            forest = RandomForestClassifier(n_estimators=10, max_depth=5, random_state=

            # Instantiate the SMOTE object
            smote = SMOTE(random_state=42)

            # Define the pipeline using make_pipeline
            grid_pipe = make_pipeline(
                StandardScaler(),
                smote,
                forest
            )

            # Create the grid parameter
            param_grid = {
                'randomforestclassifier__n_estimators': [5, 7, 10],
                'randomforestclassifier__max_depth': [5,6,7],
                'randomforestclassifier__min_samples_split': [2, 5, 7],
                'randomforestclassifier__min_samples_leaf': [1, 2, 4]
            }

            # Create the grid, with "dummy_pipe" as the estimator
            grid_search = GridSearchCV(grid_pipe, param_grid, cv=5, scoring='accuracy')

            # Fit using grid search
            grid_search.fit(X_processed_train, target_train)

            # Calculate the test score
            test_score = grid_search.score(X_processed_test, target_test)
```

```python
In [172]:   test_score
```

Out[172]:   0.7610921501706485

```python
In [173]:   confusion_mat
```
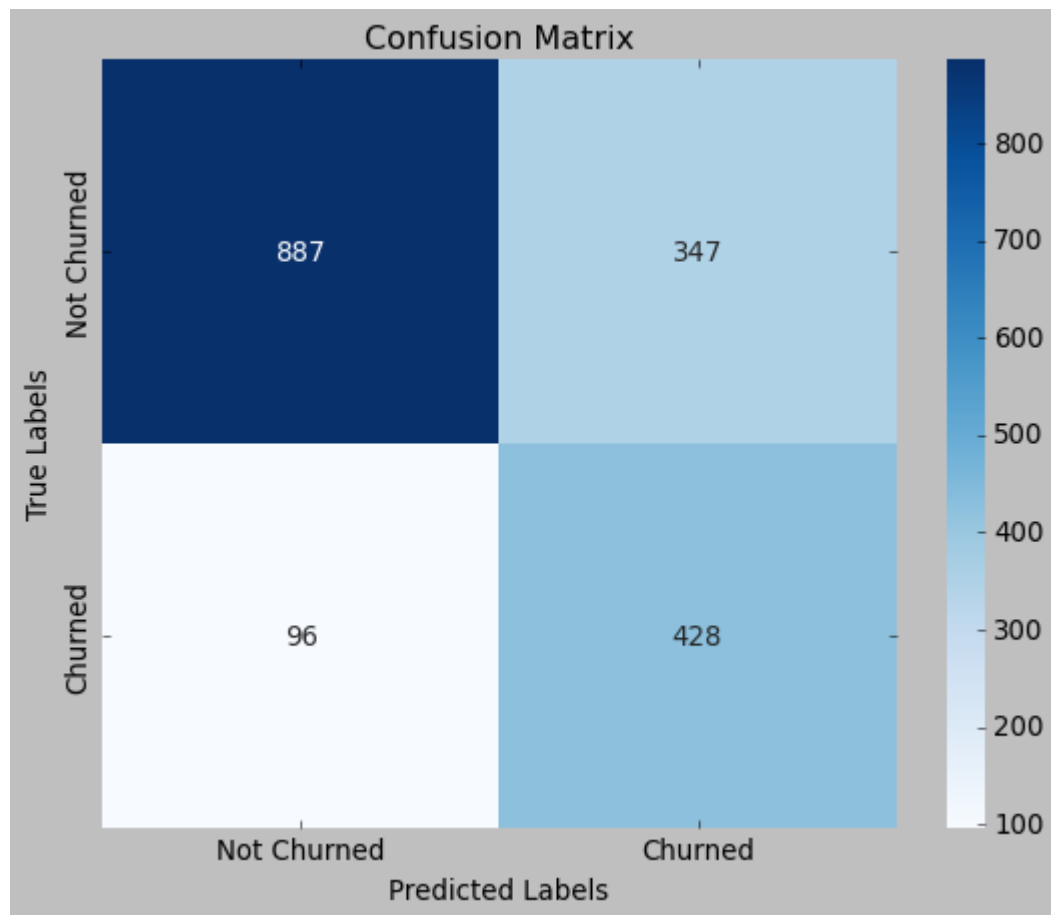
Out[173]:   array([[887, 347],
                   [96, 428]], dtype=int64)

In [174]:   ►|
```python
# Calculate the confusion matrix
confusion_mat = confusion_matrix(target_test, target_pred)

# Define class labels
class_labels = ["Not Churned", "Churned"]

# Display the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_mat, annot=True, cmap="Blues", fmt="d")
plt.title("Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.xticks([0.5, 1.5], class_labels)
plt.yticks([0.5, 1.5], class_labels)
plt.show()
```


In [175]:   ►|
```python
classify(target_test, target_pred)
```

Classification Report:

	precision	recall	f1-score	support
0	0.90	0.72	0.80	1234
1	0.55	0.82	0.66	524
accuracy			0.75	1758
macro avg	0.73	0.77	0.73	1758
weighted avg	0.80	0.75	0.76	1758

- For class 0 (not churned), a recall of 0.72 indicates that the classifier correctly identified 72% of the actual not churned instances. In other words, 77% of the customers who did not churn were correctly classified as not churned.
- For class 1 (churned), a recall of 0.82 indicates that the classifier correctly identified 82% of the actual churned instances. In other words, 82% of the customers who churned were correctly classified as churned.

V. Model Evaluation: Which is the best model?

The final iteration is the best iteration. as the confusion matrix shows an improvement in predicting the actual churned instances. In this case, we see an improvement from 50% to 80%, with only a small drop in our recall for class 0 (not churned) from 83% to 74%. This is a preferable model because it accounts for the imbalanced sample with smote. This overall is a more balanced model as it is the best prediction distribution between both classes.

Feature importances

In [176]:

```python
# Retrieve the best estimator from the grid search
best_estimator = grid_search.best_estimator_

# Access the RandomForestClassifier model within the pipeline
forest_model = best_estimator.named_steps['randomforestclassifier']

# Retrieve feature importances
importances = forest_model.feature_importances_

# Get the names of the features
feature_names = X_processed_train.columns

# Sort the feature importances in descending order
indices = np.argsort(importances)[::-1]

# Plot the feature importances as a horizontal bar chart
plt.figure(figsize=(10, 6))
plt.title("Feature Importances")
plt.barh(range(len(importances)), importances[indices], align="center")
plt.yticks(range(len(importances)), feature_names[indices])
plt.xlabel("Importance")
plt.ylabel("Features")
plt.tight_layout()
plt.show()
```
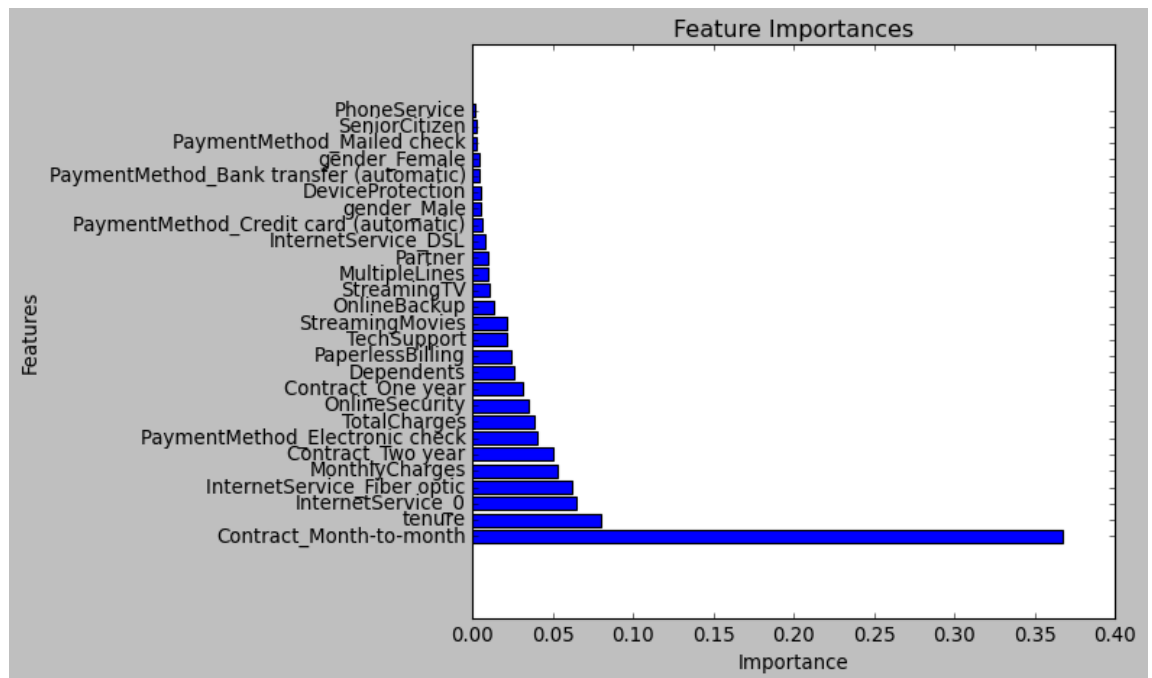

Observations

As we can see from the feature importance chart, the most important features displayed are:

- Tenure (number of months)

- Contracts (month-to-month, two year)
- Payment method of electronic checks (whether they do or do not)
- Charges (monthly and total)
- Paperless billing
- Internet Service

It is clear in the model that the tenure and whether the customers are signed up for a month-to-month contract are the most influential features, but we will observe all of them and take a look at what correlations lead to churning or not churning, and present reccomendations.

Putting dataframe back together for observation

Feature dataframe

```
In [177]: ▶  features = pd.concat([X_processed_train, X_processed_test])
```

```
In [178]: ▶  len(features)
```
Out[178]: 7032

```
In [179]: ▶  features.head()
```
Out[179]:

	gender_Female	gender_Male	InternetService_0	InternetService_DSL	InternetService_Fi op
1980	0	1	0	0	
5485	0	1	0	1	
198	0	1	0	0	
6326	1	0	0	1	
1304	0	1	0	0	

5 rows × 27 columns

Target dataframe

```
In [180]: ▶  targets = pd.concat([target_train, target_test])
```

```
In [181]: ▶  len(targets)
```
Out[181]: 7032

```
In [182]:  ▶ targets.head()
```

Out[182]: 1980   0
           5485   0
           198   0
           6326   0
           1304   1
           Name: Churn, dtype: int64

```
In [183]:  ▶ df_final = pd.concat([features,targets], axis=1)
             df_final.head()
```

Out[183]:

	gender_Female	gender_Male	InternetService_0	InternetService_DSL	InternetService_Fiber optic
1980	0	1	0	0	
5485	0	1	0	1	
198	0	1	0	0	
6326	1	0	0	1	
1304	0	1	0	0	

5 rows × 28 columns

```
In [184]:  ▶ df_final.columns
```

Out[184]: Index(['gender_Female', 'gender_Male', 'InternetService_0',
                  'InternetService_DSL', 'InternetService_Fiber optic',
                  'Contract_Month-to-month', 'Contract_One year', 'Contract_Two yea
           r',
                  'PaymentMethod_Bank transfer (automatic)',
                  'PaymentMethod_Credit card (automatic)',
                  'PaymentMethod_Electronic check', 'PaymentMethod_Mailed check',
                  'SeniorCitizen', 'Partner', 'Dependents', 'tenure', 'PhoneService',
                  'MultipleLines', 'OnlineSecurity', 'OnlineBackup', 'DeviceProtectio
           n',
                  'TechSupport', 'StreamingTV', 'StreamingMovies', 'PaperlessBillin
           g',
                  'MonthlyCharges', 'TotalCharges', 'Churn'],
                 dtype='object')

Categorical Visualization

In [185]:

```python
features = []

for i in df_final.columns:
    if i == 'Contract_Month-to-month':
        features.append(i)
    elif i == 'Contract_Two year':
        features.append(i)
    elif i == 'PaymentMethod_Electronic check':
        features.append(i)
    elif i == 'InternetService_Fiber optic':
        features.append(i)
    elif i == 'InternetService_0':
        features.append(i)

# Let's look at the distribution of the columns we have selected with a bar

plt.figure(figsize=(10, 10))

for i, column in enumerate(features):
    plt.subplot((len(features) + 1) // 2, 2, i + 1)

    sns.countplot(data=df_final, x=column, hue="Churn")

    plt.title(column, fontsize = 15, color = "red")
    plt.xticks(rotation=45)

plt.tight_layout(pad=2.0)
plt.show()
```
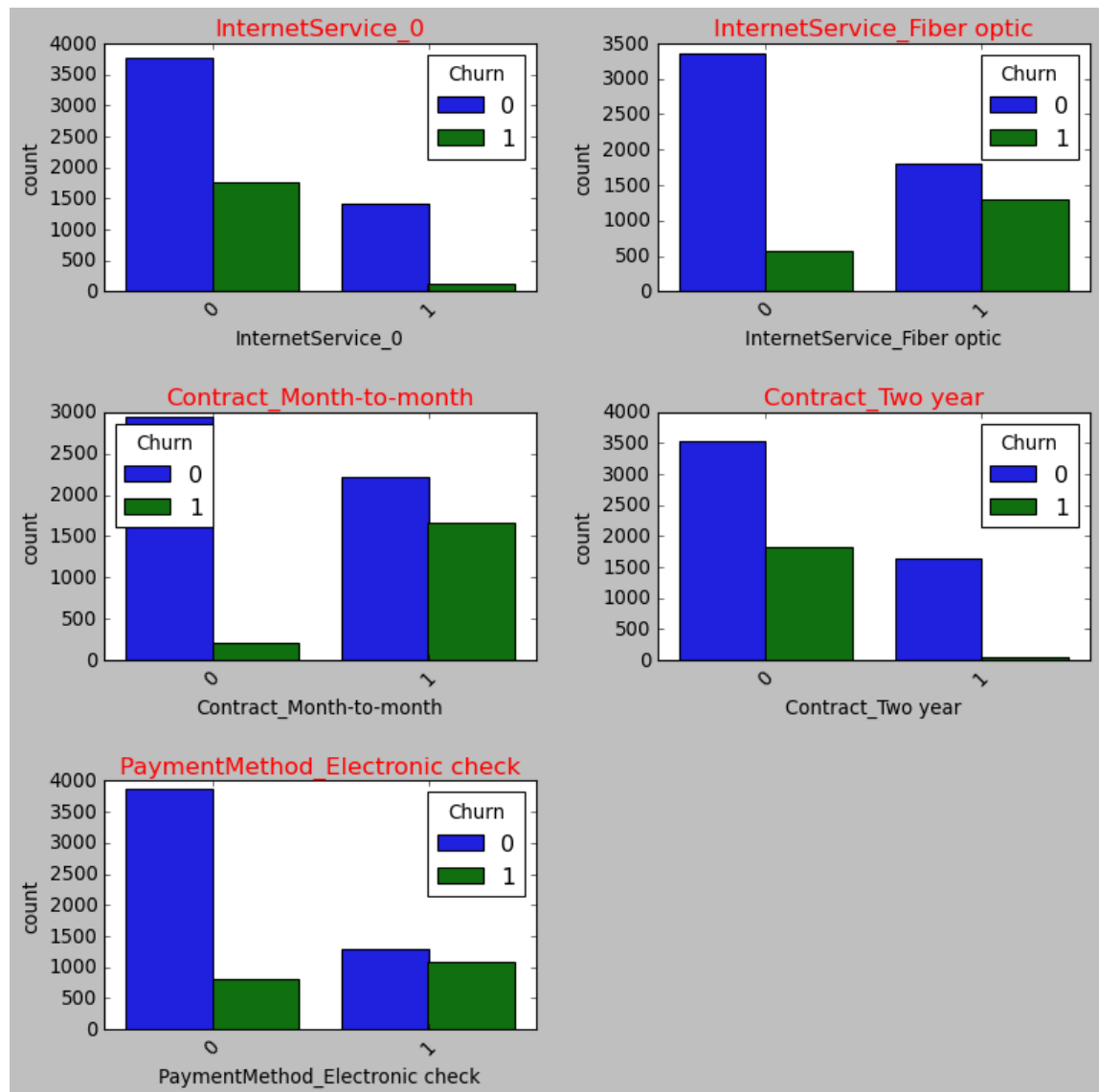
Observations

Contract Month-to-month

In the bar chart above that the contract being month to month greatly affects predictability on whether customers will or will not churn. In the bar chart, the customers who are `not signed up for month-to-month(0)` show a significantly **lower count of churning** versus if they ` were signed up month-to-month (1)` . This is likely because customers who are signed up month-to-month have an easier time of leaving Telco. Based on this chart, it would make sense to try and `minimize the number of month to month contracts` .

Contract Two-Year

In the bar chart above shows customers being signed up for a two-year contract greatly affects predictability on whether customers will or will not churn. In the bar chart, the customers who are ` not signed up for a two-year contract (0)` show a significantly **higher count of churning** versus if they ` were signed up for a two-year contract (1)` . This is likely

because customers who are signed up for a two year have more incentive to stay with the company as they are used to the service and likely don't want to change something they are already familiar with. Based on this chart, it would make sense to try and `maximize the number of two-year contracts`.

Payment Method: Electronic check

In the bar chart above shows customers being signed up for electronic checks as their payment method greatly affects predictability on whether customers will or will not churn. In the bar chart, the customers who are `not signed up for electronic checks as their payment method (0)` show a significantly **higher count of not churning** versus if they `were signed up for electronic checks as their payment method (1)`. In this chart, it is important to note that the overall distribution of feature shows that the majority of customers did not sign up for electronic checks and as a result, most of those customers did not end up churning. Based on this chart, it would make sense to `minimize customers who do sign up for electronic checks` as their preferred method of payment.

Internet Service - Fiber optic

In the bar chart above shows customers being signed up for customers being signed up for fiber optic internet service greatly affects predictability on whether customers will or will not churn. In the bar chart, the customers who are `not signed up for fiber optic internet (0)` show a significantly **lower count of churning** versus if they `were signed up for fiber optic internet (1)`. Based on this chart, it would make sense to `minimize the number of customers who sign up for fiber optic internet.`

KDE Plots of Total Charges, Monthly Charges, and Tenure

Tenure

The `tenure(# of months on contract)` is listed as the most important feature in the model. A KDE (Kernel Density Estimation) plot is a useful visualization technique for representing the distribution of a dataset. It provides a smooth, continuous estimate of the underlying probability density function (PDF) of a continuous variable. Unlike a histogram, which represents the data using discrete bins, a KDE plot offers a smoother representation of the data distribution.

A KDE plot allows you to observe the shape of the distribution. It can reveal important characteristics such as symmetry, skewness, or multimodality. By examining the shape, you can gain insights into the central tendency and spread of the data.

In [186]: ► `kde_plot(df,'Churn','tenure')`
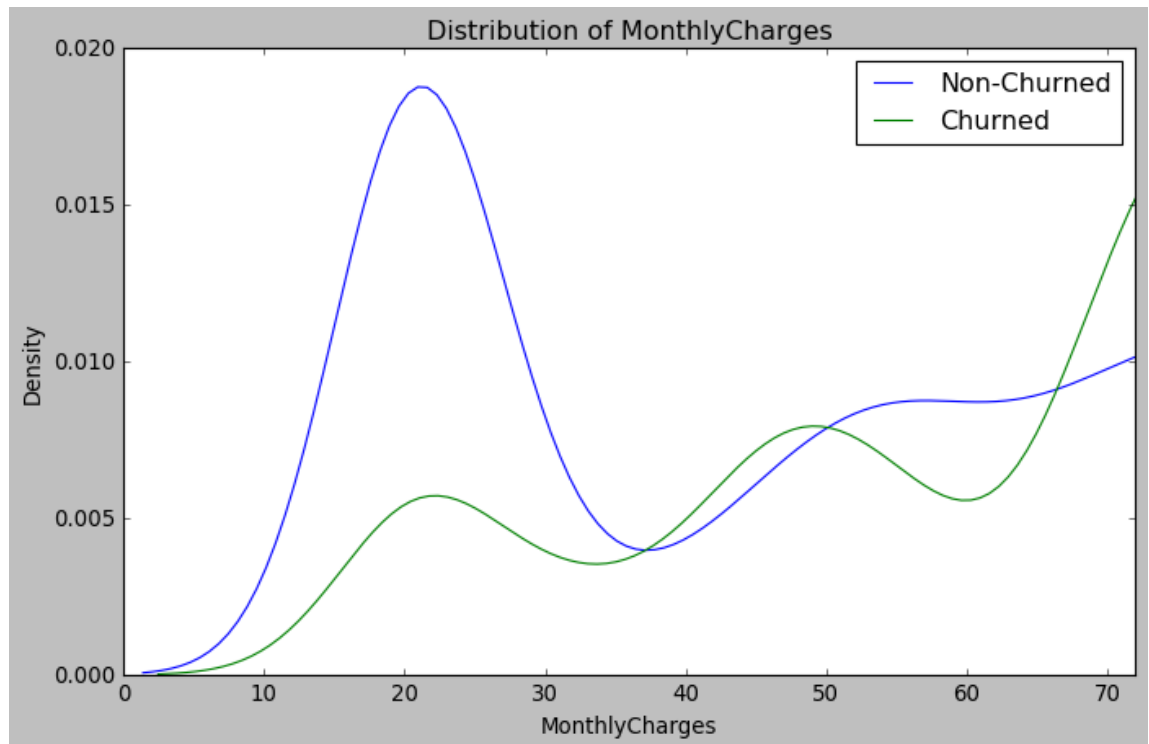

Distribution of tenure
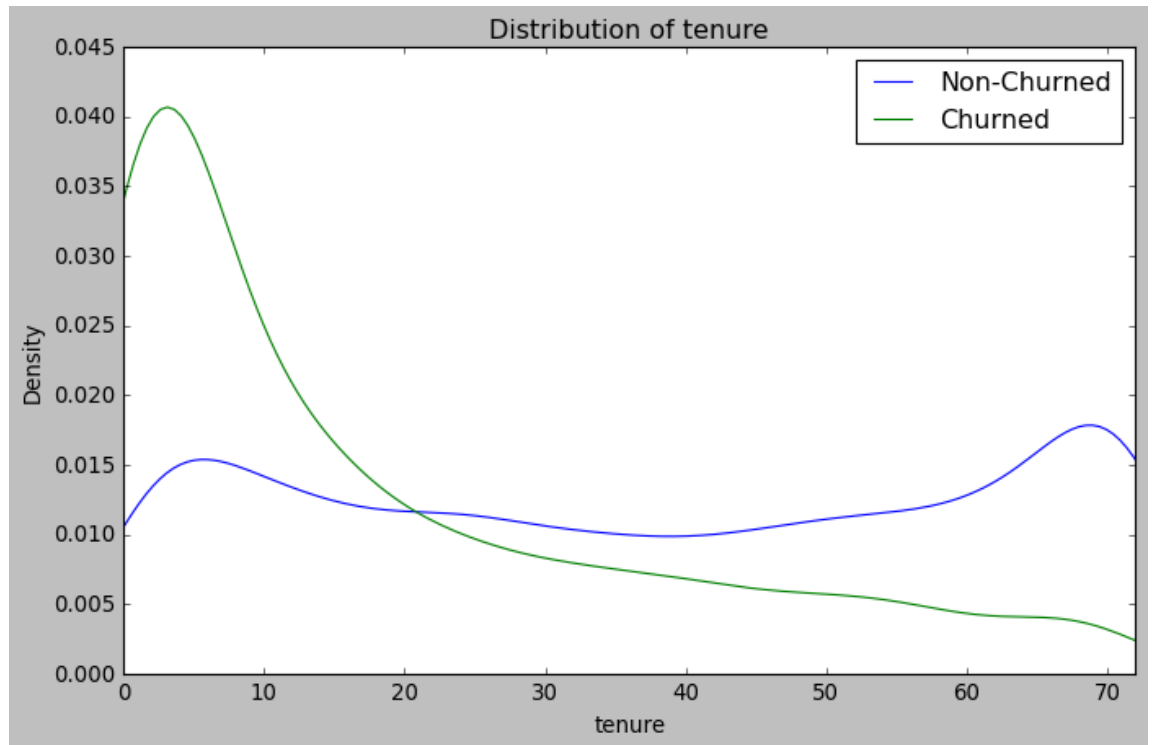
Observations

According to this kde plot, we can see that the tenure has a significant impact on churned customers. It appears that the data of churned customers in relation to tenure is most concentrated at around 0-5 months. After the customer stays with Telco for longer than 5 months, the drop off of churned customers is significant. According to this chart, it would make sense to `maximize tenure of customers` as it increases the reliability of revenue per customer. Another reccomendation would be for Telco to find discounts or additional services for initially signed up customers to keep them for those first 5 months to keep those customers from churning.

Monthly Charges

▶ `kde_plot(df,'Churn','MonthlyCharges')`

Observations

According to this kde plot, we see that the distribution of data for `non-churned customers is`
`significantly higher at a lower monthly charge of around 20 dollars` , and the
`number of churned customers is more heavily concentrated at 80-100 dollars` .
Also, it shows that `the distribution of customers who are charged monthly between`
`40-70 dollars is relatively the same for both churned and non-churned`
`customers.` With the desired effect to be to maximize non-churned customers, it would make
sense to `keep the monthly charge under 40 dollars to keep customers from`
`churning.`

Conclusion

- When finding a model for predicting the churning of a customer, the random forest classifier using gridsearch cross validation showed the best recall for learning how to maximize false negatives.
- The random forest model performed at a 76% accuracy
- For class 0 (not churned), a recall of 0.72 indicates that the classifier correctly identified 72% of the actual not churned instances. In other words, 72% of the customers who did not churn were correctly classified as not churned.

- For class 1 (churned), a recall of 0.82 indicates that the classifier correctly identified 82% of the actual churned instances. In other words, 82% of the customers who churned were correctly classified as churned.

- In this model, the features that showed of most importance were the type of contract (specifically the month-to-month vs two-year), the tenure (number of months on contract),

Recommendations

- `minimize the number of month to month contracts`
- `maximize the number of two-year contracts`
- `minimize the number of customers who use fiber optic internet.`
- `maximize tenure of customers`
- ` find discounts or additional services for initially signed up customers to keep them for those first 5 months`
- ` keep the monthly charge under 40 dollars`

Future Work

In the future new data from Telco can be observed as new services and evolvement of different options occurs. It may be useful to look at location and a dataset where age is considered as well according to the customer. More non-categorical variables like cost per customer could be good features to observe in this model to determine if the lifetime value leads to maximizing profit.