Data Mining – Customer Churn Model

Problem Statement:

Telco, a telecommunications company that offers a variety of services to their customers, has just looked over some of their customer data. They realize roughly 26% of the 5043 customers in their database have decide to end their contract with the company and defect to the competition. They would like to bring the churn rate down by 5-10%.

It is more costly for the company to lose a customer than to keep one of course (when a customer leaves, Telco loses the recurring revenues that the customer carries with them).

The board of directors are unsure what customer characteristics are most contributing to them defecting to Telco's competitors and would like a better understanding of the factors that are contributing to customer churn.

Using past data, they would like me to design a model that is able to predict propensity rates of which customers in their database are most likely to churn. Moreover, they would like me to propose solutions to the marketing and sales department to reduce overall churn rates going into the next fiscal year.

Data Selection:

Training and Validation: The data I am using for this project is the *Telco Customer Churn* dataset. It can be found here: https://github.com/KimathiNewton/Telco-Customer-Churn/blob/master/Datasets/telco_churn.csv. It is a solid dataset to use for applying classification algorithms and relevant metrics. Moreover, there are a number of features in the dataset that will need to be handled accordingly to be able to effectively perform my data mining techniques.

The dataset includes various features:

- **CustomerID**: The unique identifier (primary key) of the customer.
- **Gender**: Gender of Customer (Male or Female).
- **SeniorCitizen**: Whether or not the customer is identified as a senior citizen.
- **Partner**: Whether or not the customer has a partner.
- **Dependents**: Whether the customer has dependents or not.
- **Tenure**: Number of months the customer has stayed with the company.

- **PhoneService**: Whether the customer has subscribed to a phone service with the company.
- **MultipleLines**: Whether or not the customer has multiple phone lines with the company.
- **InternetService**: Whether the customer has subscribed to an internet service plan (Telco offers DSL and Fiber Optic)
- **OnlineSecurity**: Whether the customer has the Online Security add-on.
- **OnlineBackup**: Whether the customer subscribed to Online Backup.
- **DeviceProtection**: Whether the customer subscribed to device protection.
- **TechSupport**: Whether the customer subscribed to receive Tech Support.
- **StreamingTV**: Whether the customer subscribed to receive streaming television.
- **StreamingMovies**: Whether or not the customer subscribed to stream movies with their plan.
- **Contract**: The contract-term length of the customer (Month-to-Month, One year, Two Year).
- **PaperlessBilling**: Whether the customer has a paperless billing feature or not.
- **PaymentMethod**: How the customer pays (Electronic check, Mailed Check, Automatic Bank Transfer, or Credit Card).
- **MonthlyCharges**: The amount charged to the customer monthly.
- **TotalCharges**: The total amount charged to the customer.
- **Churn**: Whether the customer churned or not.

Data Pre-Processing:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5043 entries, 0 to 5042
Data columns (total 22 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Unnamed: 0        5043 non-null   int64  
 1   customerID        5043 non-null   object 
 2   gender            5043 non-null   object 
 3   SeniorCitizen     5043 non-null   object 
 4   Partner           5043 non-null   object 
 5   Dependents        5043 non-null   object 
 6   tenure            5043 non-null   int64  
 7   PhoneService      5043 non-null   object 
 8   MultipleLines     4774 non-null   object 
 9   InternetService   5043 non-null   object 
 10  OnlineSecurity    4392 non-null   object 
 11  OnlineBackup      4392 non-null   object 
 12  DeviceProtection  4392 non-null   object 
 13  TechSupport       4392 non-null   object 
 14  StreamingTV       4392 non-null   object 
 15  StreamingMovies   4392 non-null   object 
 16  Contract          5043 non-null   object 
 17  PaperlessBilling  5043 non-null   object 
 18  PaymentMethod     5043 non-null   object 
 19  MonthlyCharges    5043 non-null   float64
 20  TotalCharges      5038 non-null   object 
 21  Churn             5042 non-null   object 
dtypes: float64(1), int64(2), object(19)
memory usage: 866.9+ KB
```

As can be seen from an initial overview of the data upon import, a number of the features contained missing values. Moreover, a good amount of the features, that one would assume would contain integer values to represent 'yes/no', were designated as having the 'object' (string) datatype e.g., Churn.

NOTE: In my Google Colab, I used the markdown feature to label each Pre-Processing technique I applied to the data. It contains all of the code, illustration, comments etc. to assist in following this document. In this document, I will describe what was being done in my Google Colab during Pre-Processing, sequentially, and why a technique was done.

- **Dropped the 'Unnamed: 0' column from the dataset.**
    - This feature essentially mimicked the implicitly derived 'primary key' integer column that Pandas applies to each row. It was redundant; therefore, this feature was dropped accordingly.

- **Adjusted columns to be easier to work with in Python and to ensure each feature name was consistently formatted.**
    - Some variables were formatted in camelCase e.g., customerID. Other variables containing two strings were not e.g., SeniorCitizen. Moreover, some variables named with one string were capitalized e.g., Partner. Others were not e.g., gender.
    - Rather than deal with the headache of trying to remember which feature names were formatted in which way, I used a list comprehension technique to reformat each feature name into being camelCase.

- **Set the variable, 'customerID' as the Primary Key**
    - This feature's sole purpose is to uniquely identify each customer; therefore, I set it as the index in the data frame. Moreover, this helped me have a better understanding of how many descriptive variables were contained in the dataframe: 20.

- **Analyzing Variables of 'Object' Data Type That Expect to Contain Boolean or Integer Values**
    - I defined a function so that I could quickly analyze how many unique values were contained within any variable in the data frame.
        - This allowed me to discover why variables, such as 'seniorCitizen' contained more than two unique values.

```
def get_valueCounts(col):
  return df[col].value_counts()
```

```
get_valueCounts('seniorCitizen')
```

seniorCitizen	count
False	2525
0	1699
True	475
1	344

dtype: int64

- **Replacing Binary String Values with their identical Boolean values**
  - I used panda's methods to quickly replace any value of 'False' or 'No' found in the data frame with a (Boolean) False value and any 'True' or 'Yes' with a (Boolean) True value.

- **Removed the response variable from the dataset (temporarily)**
  - I used a panda's methods to remove the response variable of interest ('churn') from the data frame and temporarily store it in a 'response' variable.

- **Transformed string 0's and 1's into equivalent Boolean values.**
  - In any variable identified as being 'Categorical' throughout the data frame and containing string value representations of 0 and/or 1, these were replaced with Boolean False values for the former and Boolean True values for the latter.

- **Changed Pandas settings to only display two decimal places in the data frame**
  - A couple of the features, namely 'monthlyCharges' and 'totalCharges' variables, contained too many unnecessary integers after the decimal point.

Because these continuous values found in the data frame were describing monetary charges, I changed the python setting to strictly display no more than two values after the decimal point.

- **Corrected 'totalCharges' to be of the float datatype**
    - To do the prior step, I would need to convert 'totalCharges' from the 'object' datatype to contain all 'floats.'
        - First, I removed white space from any string value found within this feature.
        - I then put any rows containing "'" (empty) string values into a temporary variable named 'missingTotalCharges'.
        - I then dropped these rows from the original dataframe, so that I could appropriately first handle the records in the dataframe that *contained* values within the 'totalCharges' column
        - Because now all of the rows found within the 'totalCharges' column (in the original data frame) were now sure to contain string representations of floats, I applied the '.astype(float)' method to the column for data conversion.
        - I then used 'pd.concat' to merge the 'missingTotalCharges' dataframe back to the original data frame.
        - I then replaced all values of '' found within the 'totalCharges' column with 'np.nan'.

- **Handle Missing**
    - Because all of the features in the data frame were now converted to their appropriate data types and contained uniform values, it was now time to handle the missing values found within each column.
    - Because a handful of the features found within the data frame contained values dependent on those of other features (e.g., a value found within the 'multipleLines' column was highly dependent on the value found within the binary 'phoneService' column), I defined a function so that I could capture a quick birds eye view of how other features in a row looked, based on some feature containing missing values. For example:

```
def get_missing_rows(col):
  return df.loc[df[col].isnull()]
```

```
get_missing_rows('multipleLines')
```

ents	tenure	phoneService	multipleLines	internetService	onlineSecurity	onlineBackup	deviceProtection	techSupport	str
False	1	False	NaN	DSL	False	True	False	False	
False	45	False	NaN	DSL	True	False	True	True	
False	10	False	NaN	DSL	True	False	False	False	
False	1	False	NaN	DSL	False	False	True	False	
True	1	False	NaN	DSL	False	True	False	False	
...	...	...	...	...	...	...	...	...	
True	72	False	NaN	DSL	True	True	True	True	
False	10	False	NaN	DSL	False	False	False	False	

    - I then performed a query on multiple variables and discovered that if a 'parent' variable (such as 'phoneService') contained a Boolean False, its 'child' variable would either contain a null value or a string such as 'No phone service'.
    - Therefore, in all of these instances in which this relationship was discovered, I replaced the null values in the 'child' column(s) with a string value, such as 'No phone service' or 'No internet service'.

- **Creating Dummy Variables**
    - Because all categorical variables now contained values that appropriately defined each row, in a uniform manner, it was time to create dummy variables. I applied the 'get_dummies' method to the data frame, setting the 'dtype' parameter to 'int', the 'prefix_sep' parameter to '_', and the 'drop_first' parameter to True and overwrote the original data frame to contain the updated features.

- **Applying KNearest Neighbords imputation to handle the missing values in the 'totalCharges' column**
    - I realized that the 'totalCharges' column still had a handful of missing values.
    - I used my previously defined 'get_missing_rows()' function to analyze how these rows appeared in context to the other rows containing missing values and how the values of the other features appeared when these values were missing (containing 'np.nan' values).

```
get_missing_rows('totalCharges')
```

ependents	tenure	phoneService	paperlessBilling	monthlyCharges	totalCharges	gender_Male	multipleLines_True
True	0	False	True	52.55	NaN	0	0
True	0	True	False	20.25	NaN	1	0
True	0	True	False	80.85	NaN	0	0
True	0	True	False	25.75	NaN	1	1
True	0	False	False	56.05	NaN	0	0
True	0	True	True	19.70	NaN	1	0
True	0	True	False	73.35	NaN	0	1
True	0	True	True	61.90	NaN	1	1

  - At this point, I applied the 'any' method to '(df.isnull())' , just to be sure that the 'totalCharges' feature was the only column still containing missing values (it was).
  - To most accurately impute numerical float values for these missing values in the 'totalCharges' column, I decided it best to import the 'KNNImputer' function from the 'sklearn.impute' library. I applied 'KNNImpute.fit_transform' to my data frame to impute the missing values to those of its nearest neighbors.
    - The function identifies which records contained missing values for that features, and then identifies which five records are most similar to that record, based on the values of its other features.
    - The missing value is then imputed using the average of the records identified to be 'most similar' or 'nearest' to these 'missing value' records and the missing values are imputed accordingly.
  - I overwrote the original data frame to contain the appropriately imputed rows and now had a data frame in which all 5,043 records contained non-null values.

- **Joining the response variable back to the data frame**
  - Because all of the descriptive features were now wrangled appropriately, I decided to join the response variable back to the working data frame to have a 'complete' data frame of X's and the Y.

- **Dropping a record containing a null value in the response variable.**

    - After joining the response series back to the data frame, I realized that one of the records in the data frame had a missing value for the 'churn' (response) feature. Because it was only one record and there is no way to statistically determine with one-hundred percent certainty the response (churn or no-churn) for this record, I decided to drop it. I did not want to risk skewing the results of my analysis by inaccurately inferring this value.

- **Creating a binary 0/1 to represent 'no churn' and 'churn' for each record**
    - The 'churn' response variable currently contained Boolean False and True. To have these values be formatted similar to the X categorical variables, I simply converted the churn data type to be 'int' so that 0 represented False (No Churn) and 1 represented True (Churn).

- **Export Pre-Processed Data to Excel File**
    - Because I had now spent a substantial amount of time wrangling and cleaning my data, I decided to export my updated data working data frame to an excel file to store on my computer's secondary memory. This prevents me from losing my progress, just in case the Kernel running on Google Colab randomly died.

Mining Technique Selection

Now that the data was preprocessed I used code to observe the spread of each variable in tabular format.

Out of all of the predictor variables, only three of them were continuous numerical features.

- **Tenure**
    - *Range:* 0-72 months
    - *Skew:* Slightly right skewed
- **Monthly Charges**
    - *Range:* $18.40-$118.65
    - *Skew:* Slightly left skewed.
- **Total Charges**
    - *Range:* $18.80-$8,670.10
    - *Skew:* Very right skewed.

I then decided to make a 6 x 5 dimension table of boxplots of each feature, separated by 'churn' vs 'no churn' to determine if any variables could be dropped. In doing so, I decided

to drop the ‘partner’ feature because this variable showed no separation between the two classes and I could not imagine why being married vs being single should have any impact on whether a customer leaves or stays with the company. Therefore, it was dropped.

Because there weren’t too many additional predictor variables in the model and they all seemed like they could viably play a part in determining whether a customer were to leave or not leave the company, I decided to leave them all in for initial training.

Additionally, I decided to remove ‘churn’ the response at this point and store it in its own response (y) variable. The rest of the variables were stored in an X variable.

Training and Validation Split

I split 10% of the 5043 records into the validation set and kept the rest of the data to be used for training the model. Moreover, I set the random_state parameter to 0 for reproducibility and set the stratify parameter to ‘y’ to ensure that the distribution of the classes in the target variable was maintained in both the training and validation sets, because the split between the two classes in the data was weighed more heavily towards any given record belonging to the ‘No churn’ class.

Techniques

I decided to try using simpler modeling techniques at the onset of modeling to serve as a baseline comparison model going forward in predicting customer churn for the Telco dataset. Below is a summarization of each of the techniques I applied and why I chose them.

- **Logistic Regression:** Because the response variable of interest is a binary categorical feature and each of the records in the dataset belonged to one of the two classes, Logistic Regression was a natural first choice modeling technique to apply. Doing so also allowed me to infer probabilities for each record belonging to the class of interest ‘Churn’, based on a chosen cutoff threshold. Moreover, the predictions derived from this data mining technique would allow me to naturally apply a Confusion Matrix to analyze the models accuracy and error rates. From here, I could fine tune the models hyperparameters (such as the cutoff threshold) and gauge the models TPR, FPR, etc. My goal was for my model to capture as many of the True Positives as possible, whilst keeping the False Negative Rate as low as possible.

Lastly, the derived propensities would allow me to later rank the records based on probability of belonging to the class of interest and apply Lift Charts, Cumulative Gain Charts, and various other functionalities to better understand what was causing a given customer to leave or not leave the company.

- **Decision Tree:** I chose to apply a Decision Tree algorithm to the dataset, for a few reasons.
    - For one, to further assist in feature selection and perhaps make a more parsimonious model, without losing much predictive performance if I only kept the features most correlated response.
    - Visualization tool, to see a logical chain of decisions in how a machine learning technique may choose to split records into one of the two classes, based on the value of the features at any given split. Furthermore, to better understand how these features interacted with each other in distinguishing which class any given record may belong to.
    - See how compared in predictive performance relative to the other techniques applied.

- **Random Forest:** This technique builds off of bootstrap aggregation. The algorithm builds a number of decision trees using boostrapped samples of the data. Multiple trees are built in parallel; furthermore, each time a split in any of the trees is considered, that split is based on a random sample of the available features. In turn, the algorithm makes a large number of trees that look different from the next. The final predictions are based on the aggregated average predictions from all trees and the hope is that this random assortment of multiple trees will return superior predictive performance.

- **Gradient Boosting:** This is an advanced data mining technique that extends Random Forests and bagging to improve predictive performance. It constructs multiple tree models sequentially, with each tree learning from the errors of the previous ones. The algorithm focuses on reducing these errors by iteratively refining predictions. The final outcome is a collection of trees, each contributing to the prediction of the target class. Each tree's "vote" is weighted based on its accuracy, culminating in a cumulative prediction that integrates the strengths of all the individual trees.

Data Mining Results:

Below is a summary of the modeling techniques that were applied, the hyperparameter settings for each model, the strategies employed to select these settings, and the performance for each model.

Baseline Logistic Regression Model

1. ***max_iter=1000***
    a. **Purpose:** Specifies the maximum number of iterations the solver is able to run before choosing the beta coefficients that minimize the loss function.
    b. **Why *1000:*** The default value is ***100***. The parameter was set to ***1000*** simply to allow more iterations if needed.
2. ***class_weights='balanced'***
    a. **Purpose:** Because the classes in the data set are imbalanced, this parameter tells the algorithm to apply weights to the two classes.
    b. **why *balanced:*** The class weights determine to what extent the model will be penalized for making incorrect predictions for each class. By setting the parameter to *balanced*, wrong predictions on the class of interest ('churn') will carry more weight. In turn, the model treats each class more fairly and doesn't become biased towards the minority class ('No Churn').
3. ***n_jobs=-1***
    a. **Purpose:** Specifies the number of CPU cores to be used during training.
    b. **Why *-1*:** This setting uses all of the PC's available CPU cores, which speeds up the computations involved in training the model.

Results on Validation Data:

Confusion Matrix (Accuracy 0.7644)

	Prediction	
Actual	no churn	churn
no churn	277	94
churn	25	109

On the validation set, the baseline logistic regression model returned a ***76% percent accuracy rate*** and a ***24% misclassification rate***.

The ***True Positive Rate*** was ***81.34%***, meaning that the model was able to correctly classify 81.34% of the customers that did in fact leave the company.

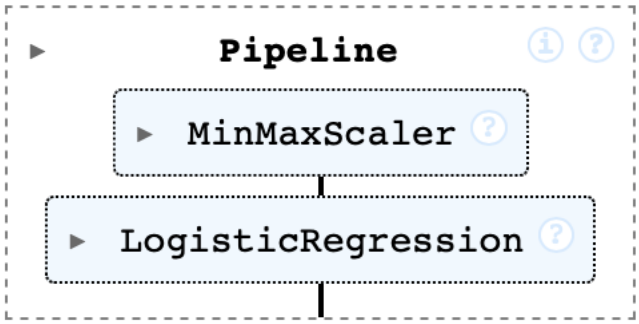
The ***False Negative Rate*** was ***18.66%***, meaning that the model predicted 'No Churn' for 18.66% of the customers who did in fact 'Churn'.

Although this performance wasn't terrible, I was confident that with some fine-tuning of the models hyperparameters, these metrics could be improved so that the model would be able to capture a greater portion of customers who churned.

NOTE: *See Appendix A for Classification Report, ROC Curve, and Precision-Recall Curve for this model.*

Baseline Logistic Regression with Min Max Scaling of Input Features

Because the predictor variables contained a couple features that were on drastically different scales relative to the others (namely, the *Monthly Charges and Total Charges*) I decided to apply a scaling technique to the inputs. Because this algorithm doesn't involve any distance calculations, I wasn't sure that this would drastically improve performance, but I felt it worth a shot. By scaling the inputs, I would be confident that each feature is contributing proportionally to the model. In doing so, I created a **Pipeline** that would first Normalize the inputs using the **Min Max Scaler** before fitting the training data to the baseline logistic regression model.



Results on Validation Data:

Confusion Matrix (Accuracy 0.7545)

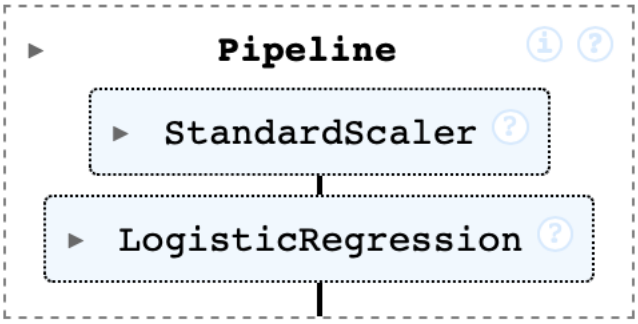
	Prediction	
Actual	no churn	churn
no churn	272	99
churn	25	109

Although the ***TPR*** and ***FNR*** did not change, the model's accuracy fell by roughly a percentage point. I wasn't confident that scaling the data would be the modification

needed to improve the predictive performance in the model; regardless, I decided to give it one more go with an alternative scaling technique.

Baseline Logistic Regression Model with Standardization

With this modification, my model would scale each input features value to a Z-score to ensure that each predictive variable contributed proportionally in determining the beta coefficients of the model.



Results on Validation Data:

Confusion Matrix (Accuracy 0.7604)

	Prediction	
Actual	no churn	churn
no churn	275	96
churn	25	109

Because the only change in performance was extremely slight degradation in accuracy, I decided, if a scaling technique was necessary, standardization would be the technique chosen.

NOTE: I went forward with Standardizing the input features of my data using the ***StandardScaler*** class from Scikit-Learn because I realized that doing so helped my model converge upon the optimal parameters. Moreover, as outlined in '*Semester Project_5.ipynb',* I did attempt strictly Standardizing the continuous features of the data, but this led to degraded performance. Therefore, I chose to Standardize all of the input features, as outlined and detailed throughout my report. For this particular dataset, this was undoubtedly the best decision for preprocessing my data.

Hyperparameter Tuning of Baseline Model (Attempt #1)

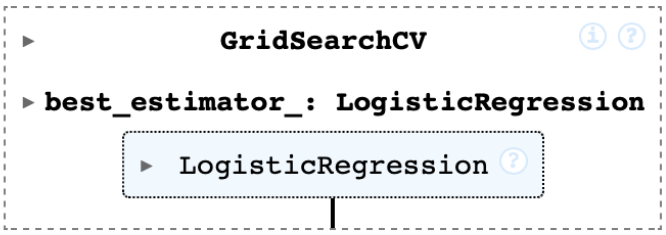
To try and increase the predictive performance of my Logistic Regression model, I made use of the ***GridSearchCV*** function from the ***sklearn.model_selection*** library. Using this function, I would be able to quickly and efficiently make multiple comparison models, set to different combinations of values for critical hyperparameters. Python would then select and return that best predictive model. Below is a code snippet of how I implemented this algorithm initially, and its results:

```
log_model = LogisticRegression(class_weight='balanced', random_state=0)
```

```
param_grid = {
    'penalty': ['l1', 'l2'],
    'max_iter': [100, 1000],
    'C': [0.01, 0.1, 1, 10],
    'solver': ['liblinear'],
    'fit_intercept': [False, True]
}
```

```
grid_log = GridSearchCV(
    log_model, param_grid, scoring='recall', cv=5
)
```

```
grid_log.fit(X_train, y_train)
```




```
grid_log.best_params_
```

Why *scoring = 'recall'*:

TelCo wants to be able to deploy a model that is able to infer which of their customers are most likely to leave the company and defect to the competition. Therefore, the best model must have a high ***recall/TPR***, even if that means accuracy is negatively impacted due to the

model incorrectly inferring whom didn't leave the company, did leave. Being able to identify which customers are leaving, and perhaps, *why* they are leaving, is TelCo's primary focus.

Why *cv= 5?* The model is essentially trained 5 times, with 5 different partitions of the training data. The data is split into 5 different subsets, each containing 20% of the training data. Each time the model is trained, the model is trained on 4 of those subsets and the remaining partition is used to validate the model. This process occurs 5 times, with each of the subsets being used once to (pre) validate the model's performance, which in turn leads to a better performing predictive model.

Best parameters returned for model:

- ***C:*** 0.1
  - ***Meaning:*** Stronger regularization penalty of beta coefficients enforced. The coefficients brought down more aggressively to 0 to reduce overfitting.
- ***fit_intercept:*** True
  - ***Meaning:*** Fits an intercept term to the logit function so that the decision boundary is shifted away from the origin to better fit the data.
- ***max_iter:*** 100
  - ***Meaning:*** Algorithm needs 100 iterations over the dataset to minimize the loss function and determine the beta's that are able to predict the class of interest with the most accuracy.
- ***penalty:*** L1
  - ***Meaning:*** Lasso regularization. Complemented by the ***C*** term, this regularization technique encourages feature sparsity by shrinking coefficients down to 0. In turn, this technique essentially performs feature selection.
- ***solver:*** liblinear
  - ***Meaning***: This loss minimization algorithm is applied to the logistic regression model to determine the best coefficients in predicting the class of interest. It is the standard, default optimization algorithm for small to medium sized datasets in sklearn for performing logistic regression.

Results on Validation Data:

Confusion Matrix (Accuracy 0.7624)

	Prediction	
Actual	no churn	churn
no churn	274	97
churn	23	111

This logistic regression model was better able to predict and identify customers who left TelCo. The ***TPR*** increased slightly (2 additional customers were identified) and the ***FNR*** decreased slightly (2 less customers were incorrectly labeled as 'No Churn' when their actual class was 'Churn'). Moreover, the ***FPR*** only increased slightly.

NOTE: *See Appendix B for a Classification Report of this model's performance.*

Hyperparameter Tuning of Baseline Model (Attempt #2)

Below is a code snippet of how I implemented this algorithm initially, and its results:

```
pipeline = Pipeline([
    ('standardScaling', StandardScaler()),
    ('lr', LogisticRegression(class_weight='balanced',random_state=0))
])
```

```
param_grid = [
    {
        'lr__solver': ['lbfgs', 'newton-cg'],
        'lr__penalty': ['l2'],
        'lr__max_iter': [100, 1000],
        'lr__C': [0.01, 0.1, 1, 10],
        'lr__fit_intercept': [False, True]
    }
]
```

```
grid_l2_log = GridSearchCV(
    pipeline, param_grid, scoring='recall', cv=5
)
```

Best parameters returned for model:

- ***C:*** 10
    - ***Meaning:*** Weaker regularization. The algorithm permits larger coefficients of the betas to better fit the model.
- ***fit_intercept:*** False
    - ***Meaning:*** Model does not add an intercept term to the logit function so that the decision boundary is forced to pass through the origin.
- ***max_iter:*** 100

    - ***Meaning:*** Algorithm uses 100 iterations over the dataset to minimize the loss function and determine the beta's that are able to predict the class of interest with the most accuracy.
- ***penalty:*** L2
    - ***Meaning:*** Ridge regularization. This discourages extremely large beta coefficients, without forcing any of the beta's to 0. This implies that most of the predictor variables have some contribution in determining whether a customer churns or doesn't churn.
- ***solver:*** lbfgs
    - ***Meaning***: This loss minimization algorithm is more efficient for medium to large datasets, ensuring faster convergence, and supports L2 regularization specifically.

Results on Validation Data:

Confusion Matrix (Accuracy 0.7149)

	Prediction	
Actual	no churn	churn
no churn	242	129
churn	15	119

Although this model's accuracy decreased by about 5%, relative to the prior run, the model's predictive performance increased in the desired metrics.

Relative to the prior model, this model was able to identify an additional 8 customers that churned from TelCo.

Relative to the baseline model, this model's ***TPR*** increased by more than ***8%***, up to an respectable ***88.81%*.**

***Because all of the desired metrics that I sought to improve, did in fact improve, I decided to move forward with this model to continue fine-tuning the hyperparameters*.**

NOTE: *See Appendix C for a ROC Curve, Precision-Recall Curve, and Classification Report for this model*.

Fine-tuning the C parameter of my current Logistic Regression model (Attempt #1)

Because my prior grid search only attempted to set values of *C (regularization strength)* to values of [0.01, 0.1, 1, 10], I decided to see if there was a value of C, perhaps in the range of [5-15] (inclusive) that may have improved my models predictive accuracy. I decided upon this range because, if the algorithm chose a value of either 5 or 10, then I could adjust my search on a subsequent run to include values in a lower or higher range.

Below is how I implemented this search:

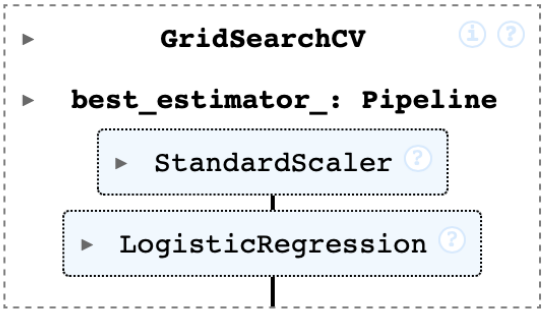
```
np.linspace(5, 15, 10)
```

```
array([ 5.        ,  6.11111111,  7.22222222,  8.33333333,  9.44444444,
       10.55555556, 11.66666667, 12.77777778, 13.88888889, 15.        ])
```

```
param_grid = [
    {'lr__C': np.linspace(5,15,10)}
]
```

```
best_C = GridSearchCV(
    pipeline, param_grid, scoring='recall', cv=5
)
```

```
best_C.fit(X_train, y_train)
```




```
best_C.best_params_
```

```
{'lr__C': 5.0}
```

Using the same hyperparameters as previously illustrated, the grid search function increased my models regularization strength when given the option to do so.

Result on Validation Data:

Confusion Matrix (Accuracy 0.7188)

Actual	Prediction no churn	churn
no churn	244	127
churn	15	119

Because this model's **TPR and FNR** remained identical to the prior run, but saw a slight improvement in **TNR** (the model was able to better distinguish 'Non Churners' from 'Churners'), I decided to move forward with the C parameter set to 5.

Fine-tuning the C parameter of my current Logistic Regression model (Attempt #2)

Below is how I continued attempting to fine tune the setting of the C hyperparameter of my logistic regression model (using all other settings previously illustrated), and the value returned:

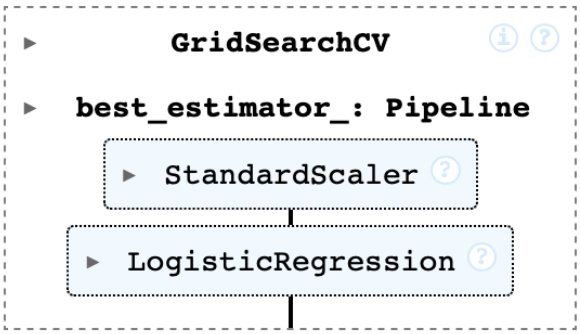
```
np.linspace(2.5, 7.5, 10)
```

```
array([2.5       , 3.05555556, 3.61111111, 4.16666667, 4.72222222,
       5.27777778, 5.83333333, 6.38888889, 6.94444444, 7.5       ])
```

```
param_grid = [
    {'lr__C': np.linspace(2.5, 7.5, 10)}
]
```

```
best_C_2 = GridSearchCV(
    pipeline, param_grid, scoring='recall', cv=5
)
```

```
best_C_2.fit(X_train, y_train)
```




```
best_C_2.best_params_
```

```
{'lr__C': 3.0555555555555554}
```

Result on Validation Data:

Confusion Matrix (Accuracy 0.7188)

Actual	Prediction: no churn	Prediction: churn
no churn	244	127
churn	15	119

No change in classification performance observed.

NOTE: *See Appendix D for a Classification Report of this model.*

Incorporating Repeated Stratified K-Fold sampling into Grid Search algorithm to observe if any potential 'subjective' hyperparameters in my current Logistic Regression Model are adjusted

Below is how I implemented my adjusted pipeline and Grid Search to incorporate Repeated Stratified K-Fold sampling of the training data to fit the model:

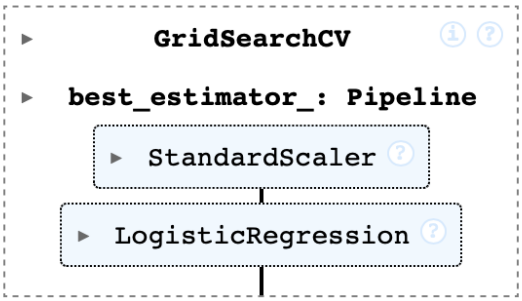
```
from sklearn.model_selection import RepeatedStratifiedKFold
```

```
pipeline = Pipeline([
    ('standardScaling', StandardScaler()),
    ('lr', LogisticRegression(class_weight='balanced', max_iter=100,
                              penalty='l2', solver='lbfgs'))
])
```

```
param_grid = [
    {'lr__C': np.linspace(2.5, 7.5, 10),
     'lr__fit_intercept': [False, True]}
]
```

```
new_model = GridSearchCV(
    pipeline, param_grid, scoring='recall',
    cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=10, random_state=0)
)
```

```
new_model.fit(X_train, y_train)
```



Why set *cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=10, random_state=0)*?

- **Repeated Stratified K Fold sampling:** builds upon Stratified K-Fold (cross validation) of proportioning the training sample to fit the model by repeating the cross-validation process multiple times with different splits. This will provide a more robust estimate of model performance by averaging the results over multiple iterations of K-Fold cross validation splits.
- **Why *n_splits=5?*** Each Stratified K-Fold iteration will contain 5 equal sized splits of the data, with each one being left out for validation each time the model is fit to the remaining 80% of the data.
- **Why *n_repeats=10?*** This carries out 10 iterations (repeats) of the above outlined Stratified K-Fold process. Upon each repeat, the Stratified samples will each be made up of now customers. This increases the robustness of the model (reducing variability), because the model is being fit to 50 different samples and subsequently validated to 50 samples of the data.

Resulting parameters:

```
{'lr__C': 3.611111111111111, 'lr__fit_intercept': False}
```

We see that the model remined true in deciding to leave out an intercept term in the logit function for best classification performance. Moreover, the C penalty term was decreased slightly.

Result on Validation Data:

Confusion Matrix (Accuracy 0.7188)

	Prediction	
Actual	no churn	churn
no churn	244	127
churn	15	119

No change in classification performance; however, because the C parameter was adjusted slightly, I decided to move forward with these hyperparameter settings being incorporated into my logistic regression model. It is reasonable to suspect that this grid search algorithm returned the C result that was appropriate for my desired performance.

Attempt in incorporating Interaction and Polynomial terms of features into current Logistic Regression model

I decided to test, specifically, if the model's performance could be enhanced by introducing interaction between the continuous predictor variables in the data. My logic was, perhaps if say, both the ***tenure*** and ***total charges*** of a customer is either low or high, this has a greater significance in whether a customer will churn or not churn. To best capture this potential relationship between the (few) continuous features in the model and the response, I introduced ***interaction (multiplicity)*** between these features to incorporate additional features in this version of my model.

Moreover, in case a given continuous predictive input on its *own* has a greater effect (and potentially non-linear relationship) on the response of each customer (churn/no churn), I decided to attempt including ***polynomial features*** of each of the continuous variables as well.

Below is how I implemented these features into a new model:

NOTE: I simplified the hyperparameters of my Logistic Regression model because, with the prior settings (e.g., ***penalty = 'L2'***), the model could not converge on an optimal solution during fitting

```
from sklearn.preprocessing import PolynomialFeatures
```

```
polyFeatures = pd.DataFrame(PolynomialFeatures(degree=2, include_bias=False).fit_transform(
    df[['tenure', 'monthlyCharges', 'totalCharges']]),
    columns=['tenure','monthlyCharges','totalCharges','tenure_squared','tenure_monthlycharges',
             'tenure_totalCharges','monthlyCharges_squared','monthlyCharges_totalCharges',
             'totalCharges_squared'], index=df.index).drop(['tenure','monthlyCharges','totalCharges'],axis=1)
 #tenure, monthlyCharges, totalCharges, tenure**2, tenure*monthlyCharges, tenure*totalCharges, monthlyCharges**2,
   # monthlyCharges*totalCharges, totalCharges**2
```

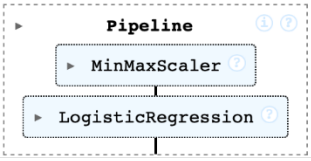
```
df_PolyFeatures = df.join(polyFeatures, how='inner').sort_index(axis='columns')
```

```
X = df_PolyFeatures.drop(['churn','partner'],axis=1)
y = df_PolyFeatures.churn
```

```
X_train, X_valid, y_train, y_valid = train_test_split(
    X, y, test_size=0.1, stratify=y, random_state=0
)
```

```
minmax_base = Pipeline([
    ('minmax', MinMaxScaler()),
    ('lr', LogisticRegression(class_weight='balanced', max_iter=1000, n_jobs=-1, random_state=0))
])
```

```
minmax_base.fit(X_train, y_train)
```



Why *degree = 2*? This returns every combination of the features less than or equal to the degree. I chose to analyze the performance of this setting, to see how increasing the

complexity of the training data impacted performance (without increasing complexity too drastically, e.g., setting *degree = 3*).

Why *bias = False*? This prevents the function from introducing a bias term into my data frame.

NOTE: As shown in the script above, interaction and polynomial transformation was only introduced to the three continuous features in my data set. The comments above in my script (paired with the names of each new column in the data frame) illustrate the resulting features that were returned by the *PolynomialFeatures* function and how these features were effectively merged together with the categorical features. Training and validation subsequently followed.

Result on Validation Data:

```
Confusion Matrix (Accuracy 0.7564)

          Prediction
  Actual no churn    churn
no churn      277       94
   churn       29      105
```

The **TPR** and **FNR** of my model, both critical to TelCo's need in reliably identifying customers most likely to churn, were both negatively impacted. Therefore, I decided introducing interaction and polynomial features into the model was unnecessary.

NOTE: *Please see Appendix E to see the validation results when introducing interaction and polynomial features (as above) with the input features being Standardized rather than Normalized (the above Pipeline has the data scaled using the MinMax scaler). Results are similar.*

Feature Selection incorporation into 'Best Logistic Regression' Pipeline

Using Feature Selection, I would instruct my algorithm to cast aside the inclusion of features identified as having low variance. Because, this insinuates that these features likely aren't informative as they have mostly the same value throughout the data set; therefore, they couldn't possibly be used to distinguish whether a customer churns or doesn't churn from TelCo.

Below is how I incorporated this technique into my 'best logistic regression' pipeline I had been slowly constructing:

```
from sklearn.feature_selection import VarianceThreshold
```

```
variable_selection = Pipeline([
    ('feature_selection', VarianceThreshold(threshold=0.15
                                            )),
    ('standardScaling', StandardScaler()),
    ('lr', LogisticRegression(class_weight='balanced', fit_intercept=False, max_iter=100,
                              penalty='l2', solver='lbfgs', C=3.6111111 ))
]).fit(X_train, y_train)
```

Why *threshold = 0.15*? This sets the minimum variance threshold for each feature. By setting the value equal to 0.15, the algorithm will only keep features whose variance is greater than or equal to 0.15.

Features identified as having low variance:

2	paymentMethod_Credit card (automatic)	True
3	paymentMethod_Mailed check	True
4	seniorCitizen	False
5	streamingMovies_No internet service	True
6	streamingTV_No internet service	True
7	techSupport_No internet service	True
8	techSupport_True	True
9	tenure	True

Because only ***seniorCitizen*** was identified as having low variance, I decided not to move forward with this technique. It is logical to suspect that a customer in the overall population of the data being a senior citizen *does* have an effect on whether that customer leaves or stays a customer with a telecommunications provider (that being TelCo).

Feature Importance

I imported the ***DecisionTreeClassifier*** function, to make a decision tree object, to use of the ***feature_importances_*** method and explicitly gauge which features in the data were identified as being informative in determining whether a customer churns or not.

Background on Decision Trees: This data mining technique recursively splits the data. The splits are made using a value within each feature to determine what is the optimal split in the data. If a feature is higher up in the tree (or even at the *root*) then this feature is identified as being *more significant*. In other words, by splitting the data into two groups by some cutoff value within this feature, the model is able to best separate the records into one of the two classes. As such, those features that appear higher up in the tree and/or appear more frequently in the tree, are identified as being *more significant* to the model's overall ability in labeling a customer as 'churn' or 'no churn'.

Resulting output

NOTE: *Higher values within the 'importance' column are identified as being more significant in the model's ability to classify. Therefore, I sorted the features in descending order, based on the paired values found within this column..*

Most Significant Features:

```
pd.DataFrame([(col, coef) for col, coef in zip(X_train.columns, dt.feature_importances_)],
             columns=['feature','importance']).sort_values('importance', ascending=False, ignore_index=True)
```

	feature	importance
0	totalCharges	0.207136
1	monthlyCharges	0.195944
2	tenure	0.189941
3	internetService_Fiber optic	0.115619
4	gender_Male	0.029130
5	paperlessBilling	0.022670
6	seniorCitizen	0.022261
7	onlineBackup_True	0.022073
8	multipleLines_True	0.020842
9	contract_Two year	0.020323
10	dependents	0.018319
11	contract_One year	0.017214
12	paymentMethod_Electronic check	0.016973
13	deviceProtection_True	0.016122
14	streamingMovies_True	0.014927
15	onlineSecurity_True	0.014655
16	techSupport_True	0.011830
17	paymentMethod_Mailed check	0.010056
18	paymentMethod_Credit card (automatic)	0.009791

Least Significant Features:

19	streamingTV_True	0.009306
20	streamingTV_No internet service	0.006236
21	deviceProtection_No internet service	0.005693
22	multipleLines_No phone service	0.002183
23	techSupport_No internet service	0.000758
24	onlineBackup_No internet service	0.000000
25	streamingMovies_No internet service	0.000000
26	onlineSecurity_No internet service	0.000000
27	phoneService	0.000000
28	internetService_DSL	0.000000

Analysis and Business Interpretation: Concerning the least significant features in determining customer churn, there are some notable observations to gain. For one, both of Telco's ***Phone Service*** and ***DSL Internet Service*** packages were not used at all by the model to classify a customer as 'churn' or 'no churn'. This leads me to the impression that neither of these features have any relationship with the response of TelCo's data mining task, meaning that neither of these services are causing any sort of effect on a customer leaving TelCo.

Moreover, all of the features with the prefixed '_No internet service' column names are viewed as redundant by the algorithm. This implies that all of the information needed to determine customer churn is obtained from these features 'parent feature'. For example, if ***OnlineBackup*** is ***False***, meaning a customer does not have the Online Backup service offered by TelCo, then the model does not gain any predictive information from a variable that states that customer doesn't have internet service either. The model already gains the information of whether a customer has internet service from variables such as ***Fiber Optic*** and ***DSL***.

Attempt at removing the five 'least significant' features from Logistic Regression model, to see how predictive performance is impacted

Because the ***Feature Selection*** identified five variables from the data as having *no* effect on the model being able to distinguish whether a customer 'churns' or not from Telco (in other words, a tree allowed to grow to max depth and completely overfit the data never made use of these features even once), I decided to try dropping them from the dataset.

Below shows the coding techniques I used to implement my procure and how I adjusted my grid search and pipeline for the Logistic Regression model to gauge performance after fitting:

```
X = df.drop(['churn','partner','internetService_DSL',
             'onlineSecurity_No internet service', 'streamingMovies_No internet service',
             'onlineBackup_No internet service','phoneService'], axis=1)
y = df.churn
```

```
X_train, X_valid, y_train, y_valid = train_test_split(
    X, y, test_size=0.1, stratify=y, random_state=0
)
```

```
param_grid = [
    {'lr__C': np.linspace(2.5, 7.5, 10),
     'lr__fit_intercept': [True, False]}
]
```

```
pipeline = Pipeline([
    ('standardScaling', StandardScaler()),
    ('lr', LogisticRegression(class_weight='balanced', max_iter=100,
                              penalty='l2', solver='lbfgs'))
])
```

```
model = GridSearchCV(
    pipeline, param_grid, scoring='recall', cv=5
)
```

```
model.best_params_
```

```
{'lr__C': 5.277777777777778, 'lr__fit_intercept': False}
```

Result of model on Validation Data:

Confusion Matrix (Accuracy 0.7168)

	Prediction	
Actual	no churn	churn
no churn	243	128
churn	15	119

NOTE: *Please see Appendix F for Classification Report of this model.*

Model Consideration: Because the models predictive performance was more or less the same relative to my prior best identified Logistic Regression model (‘using all predictors other than ***partner***), this model is certainly worth considering. However, because there aren’t too many features in this dataset to begin with and some features, such as ***internetService_DSL*** *could* at some point be a determinant in whether a customer churns from TelCo, I have decided to keep the features in the model, at least for the time being.

Keeping them in the model doesn't seem to impact performance; however, it is worth noting their lack of contribution to the model's performance and noting the reasons above as to why.

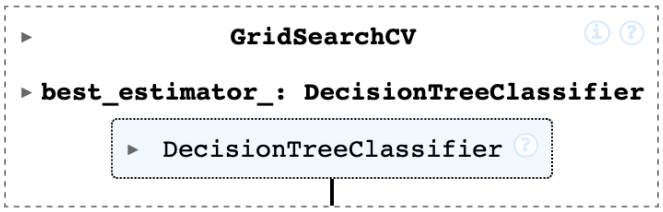
Decision Tree

Below is my code script for how I implemented this data mining technique. In implementing a decision tree model, I wanted to compare how a decision tree recursively splitting the customers into groups, based on the values found within each distinct feature, would compare with the my best logistic regression model:

```
param_grid = {
    'max_depth': list(range(1, 26, 5)),
    'min_samples_split': list(range(2, 52, 3)),
    'min_impurity_decrease': [0, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1, 0.5]
}
```

```
gridSearch = GridSearchCV(DecisionTreeClassifier(random_state=0),
                          param_grid, scoring='recall', cv=5, n_jobs=-1)
```

```
gridSearch.fit(X_train, y_train)
```



Here were the hyperparameters selected for the tree using Grid Search Cross Validation:

```
{'max_depth': 16, 'min_impurity_decrease': 0.0005, 'min_samples_split': 44}
```

Why *max_depth set to 16*? This means that the longest path of the tree consists of 16 nodes (1 root node, 15 decision nodes, and 1 leaf node). This prevents the tree from overfitting and making splitting decisions based on random noise in the data that is not indicative of whether a customer churns or not. Furthermore, the tree will be able to better generalize to unseen data.

Why *min_impurity_decrease set to 0.0005*? At each node, a split is made based on impurity decrease. For example, at a decision node, if the tree finds that customers all having ***tenure*** with TelCo less than 6 months belong to 'churn' and all customers with ***tenure*** greater than or equal to 6 months don't churn, then this split will be chosen. In fact, this would be the only feature needed by the tree to make its classifications. However, such a feature is rarely ever found, so each significant feature that can best separate the two classes is used, until the tree is grown to maximum length (and overfit) or it is told to stop with some chosen criteria set by the user. Here, to compliment the max_depth hyperparameter, this tells the algorithm that a split can only be made if the split decreases the impurity by a value of 0.0005. For a binary classification task, such as TelCo's, a gini-value (the default impurity in Python's sklearn library) of 0 would represent a terminal (leaf/end) node in which all the data points belong to one class. A value of 1 would represent a node in which there is a perfect 50/50 mix of both customers who churn and don't churn. Therefore, this parameter helps prevent the tree from overfitting by reducing the number of splits the tree will make.

Why *min_samples_split set to 44*? This hyperparameter requires that a node can contain no less than 44 customers at any given split. This parameter, similarly, prevents the tree from overfitting.

Results on the Validation Data:

Confusion Matrix (Accuracy 0.7921)

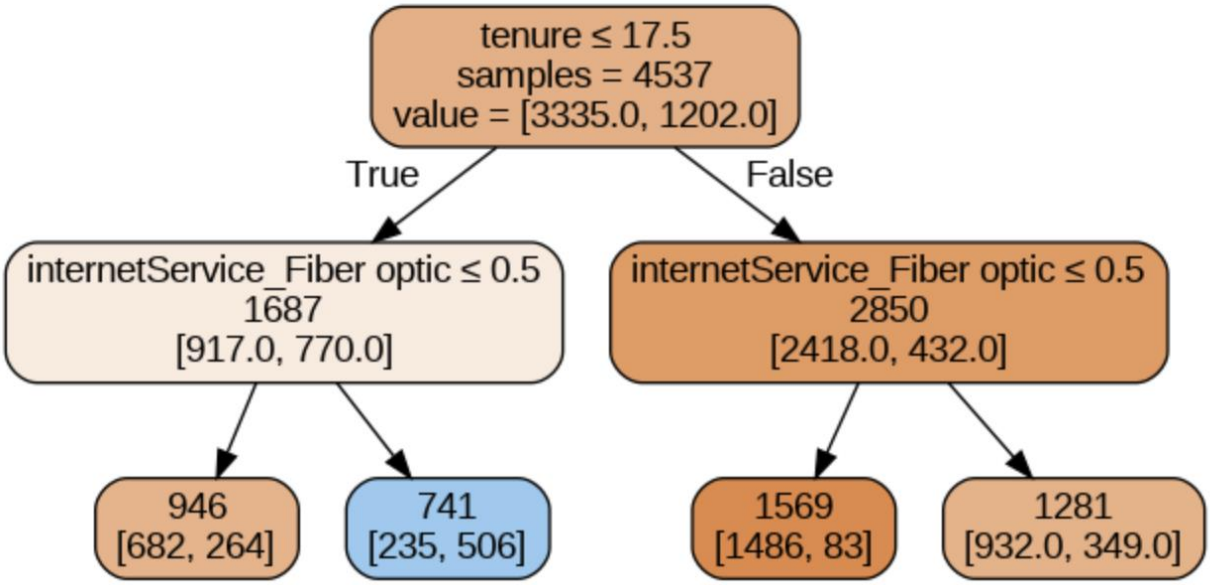
	Prediction	
Actual	no churn	churn
no churn	318	53
churn	52	82

Although the tree had better overall accuracy in predicting the two classes, it's ability to identify the class of interest in the data (customer who churned) fell off drastically.

Visualization of some the most important features and their interaction, using the Decision Tree

Using the same predefined parameter grid for the Grid Search function to determine the trees hyperparameters, I imported the ***plotDecisionTree*** function from the ***dmba*** library to explicitly see how the tree split the data at the top level. Doing so allowed me to better understand what combination of variables were most contributing to customer's churning from TelCo.

Results on Training Data:



Of the 741 customers in the training data who have been identified as 'doing business' with Telco for less or equal to 17.5 months *and* having subscribed to their Fiber Optic internet service package, 506 of them left TelCo. We also see that, of the 1,569 customers of TelCo's who have been with the company *longer* than 17.5 months and *don't* have the Fiber Optic internet service package, only 83 of them churned from TelCo. The other 1,486 customers have remained loyal customers.

Business insight: The Fiber Optic internet service package seems to be a *major* indicator of whether a customer remains with TelCo or doesn't. Having this package seems to greatly increase the likelihood that a customer will distance themselves from doing business with TelCo altogether. Moreover, it appears that customers who *do* have Fiber Optic IP from TelCo, but have been with the company longer than 17.5, does not in any way determine whether a customer leaves or not. This could be because these customers bought the service at a time when the service provided faster download speeds and was meeting customers expectations. Perhaps, the download and upload speed of the Fiber Optic service TelCo offers falls shorts of the expectations of new customers who have recently switched to TelCo, from another internet service provider. As such, this service needs to be properly analyzed and reconfigured before it is marketed towards TelCo's customers.

Random Forest

Below is how I implemented the data mining technique:

```
X = df.drop('churn', axis=1) # including partners
y = df.churn
```

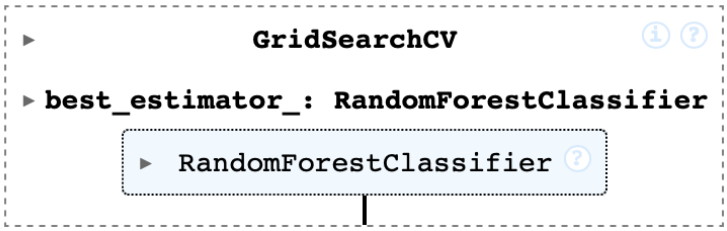
```
X_train, X_valid, y_train, y_valid = train_test_split(
    X, y, test_size=0.1, stratify=y, random_state=0
)
```

```
param_grid = {
    'max_depth': [4, 8, 12, 16, None],
    'min_samples_leaf': [1,4, 6, 10, 14],
    'min_impurity_decrease': [0, 0.0001, 0.0005, 0.001, 0.005, 0.1, 0.5]
}
```

```
rf = RandomForestClassifier(random_state=0)
```

```
rf_grid = GridSearchCV(
    rf, param_grid, cv=5, scoring='recall', n_jobs=-1
)
```

```
rf_grid.fit(X_train, y_train)
```



Best Parameters

```
{'max_depth': 12, 'min_impurity_decrease': 0, 'min_samples_leaf': 1}
```

Results on Validation Data:

```
Confusion Matrix (Accuracy 0.7960)

          Prediction
  Actual no churn    churn
no churn      328       43
   churn       60       74
```

Because this algorithm is said to perform best when each tree is allowed to grow to full length, I left this option possible in my parameter grid for the model. Moreover, I left each feature in the training data so that the algorithm could truly discern and learn, at random, what the most effective features were in the data. Regardless, the algorithm severely undershot my expectations. My thought is that it is because there *is* indeed a clear major set of feature interactions that are causing a customer churn, and an ensemble of random trees built in parallel cannot improve the classification accuracy when this interaction is so definite. Moreoever, because the output of this model is not as interpretable as those from a logistic regression or single decision tree model, I did not bother much more in figuring out exactly why this was.

Gradient Boosting

Below is how I attempted to implement this technique into my overall machine learning pipeline:

```
gb = GradientBoostingClassifier(
    random_state=0
)
```

```
param_grid = {
    'max_depth': [4, 8, 12, 16, None],
    'min_samples_leaf': [1, 2, 4, 6],
    'learning_rate': [0.1, 0.5, 1]
}
```

```
gb_grid = GridSearchCV(
    gb, param_grid, cv=5, scoring='recall', n_jobs=-1
)
```

Why *learning_rate:* This hyperparameter defines each trees contribution to the overall classification process. A smaller value (less than 1) has each tree learn more slowly, requires more trees, and generally increases generalization to validation data, whilst larger

values speed up the learning process but increases the likelihood that the tree will overfit or overstate the minimum loss in prediction.

Result:

```
gb_grid.fit(X_train, y_train) # took too long.. was on 15 minutes and still hadn't converged
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-33-2be24f952ba2> in <cell line: 1>()
----> 1 gb_grid.fit(X_train, y_train)

                                 7 frames
/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in _retrieve(self)
   1760                 (self._jobs[0].get_status(
   1761                     timeout=self.timeout) == TASK_PENDING)):
-> 1762                 time.sleep(0.01)
   1763                 continue
   1764

KeyboardInterrupt:
```

As illustrated above, after a fifteen-minute wait, the tree still hadn't converged and was unable to fit to the training data. My thought is I may have potentially provided the algorithm with too small of values to choose from for the ***min_samples_leaf*** hyperparameter. Regardless, because I knew that this model could not lend much interpretive reasoning to my analysis and help me zoom in on the overall issue that was causing customer's to churn from TelCo, I decided to move on.

Voting

To gauge how my best logistic regression model and best random forest ensemble model may perform as a 'team,' I decided to try and implement a voting technique into my pipeline. In this way, I could combine the inferences of both techniques to perhaps arrive to a superior and robust classification decision.

To do so, I made use of the ***VotingClassifier*** function to aggregate both models predictions on classifying customer churn.

Below is how I implemented this approach:

```
lr_pipeline = Pipeline([
    ('scaling', StandardScaler()),
    ('lr', LogisticRegression(class_weight='balanced', fit_intercept=False, max_iter=100,
                              penalty='l2', solver='lbfgs', C=3.611111))
])
```

```
lr_grid = GridSearchCV(
    lr_pipeline, param_grid={}, scoring='recall', cv=5
)
```

```
rf_pipeline = Pipeline([
    ('scaling', StandardScaler()),
    ('rf', rf_grid.best_estimator_)
])
```

```
from sklearn.ensemble import VotingClassifier
```

```
majority_rules = VotingClassifier([
    ('lr', lr_grid),
    ('rf', rf_pipeline)
], voting='hard').fit(X_train, y_train)
```

```
max_probabilities = VotingClassifier([
    ('lr', lr_grid),
    ('rf', rf_pipeline)
], voting='soft', weights=[4,1]).fit(X_train, y_train)
```

Why set *voting = 'hard'*? This ensures that the final predictions are based on 'majority rules'. It's worth noting that, because I only used two models, the effect of this classification technique wasn't as robust and definitive as it would have been had I implemented a third model.

Why set *voting = 'soft'*? This tells the algorithm to determine the prediction of the classes based on the average of all propensities across classes. The highest average propensity determines class prediction.

Why set *weights = [4, 1]*? This tells the algorithm to place more emphasis on the predicted probabilities from the logistic regression model and less emphasis on those from the random forest model. This was done because, when the random forest was given more equal weighting (say) in casting predictions, the models performance was negatively impacted.

Result on Validation Data:

Hard Voting

Confusion Matrix (Accuracy 0.8079)

Actual	Prediction: no churn	Prediction: churn
no churn	333	38
churn	59	75

Soft Voting

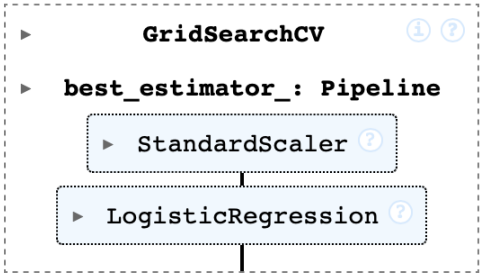
Confusion Matrix (Accuracy 0.7406)

Actual	Prediction: no churn	Prediction: churn
no churn	260	111
churn	20	114

Because the model only performed reasonably well when a *major* weighting skew was given to those cast by the logistic regression model, I decided to not forward with this technique.

Best Model up to this point: Logistic Regression Model with Standardized input features and Fine-Tuned Hyperparameters

```
lr_grid.fit(X_train, y_train)
```




```
classificationSummary(y_valid, lr_grid.predict(X_valid), class_names=class_names)
```

```
Confusion Matrix (Accuracy 0.7188)

         Prediction
  Actual no churn    churn
no churn      244      127
   churn       15      119
```

Trying New Cutoff Threshold for my Logistic Regression Model to Increase TPR and decrease TNR

Below is how I began implementing this approach, using the default class predictions in which the model predicts 1 (churn) if predicted probability of belonging to churn is greater than or equal to 0.5 and predicts 0 if less than 0.5.

```
df = pd.DataFrame({
    'predicted': lr_grid.predict(X_valid),
    'actual': y_valid,
    'prob_churn': lr_grid.predict_proba(X_valid)[:,1]
})
```

```
df.sort_values(by='prob_churn', ascending=False, inplace=True)
```

```
df.head()
```

customerID	predicted	actual	prob_churn
7024-OHCCK	1	1	0.954586
4695-WJZUE	1	1	0.942162
8884-ADFVN	1	1	0.941698
2446-BEGGB	1	1	0.941602
1724-BQUHA	1	1	0.940507

I then defined a function so that I could quickly see how my logistic regression model performed across different cutoff thresholds:

```
def classSummary(data, class_summ, cutoff, class_names):
  predicted = [1 if p > cutoff else 0 for p in data.prob_churn]

  class_summ(data.actual, predicted, class_names=class_names)
```

Here is the performance across viable thresholds for the metrics I aimed to maximize (TPR):

```
# cutoff = 0.5

classSummary(df, classificationSummary, 0.5, ['no churn', 'churn'])
```

```
Confusion Matrix (Accuracy 0.7188)

         Prediction
  Actual no churn    churn
no churn      244      127
   churn       15      119
```

```
#cutoff = 0.4

classSummary(df, classificationSummary, 0.4, class_names)
```

```
Confusion Matrix (Accuracy 0.6535)

         Prediction
  Actual no churn    churn
no churn      203      168
   churn        7      127
```

```
#cutoff = 0.45

classSummary(df, classificationSummary, 0.45, class_names) #ideal threshold I believe
```

```
Confusion Matrix (Accuracy 0.6931)

         Prediction
  Actual no churn    churn
no churn      224      147
   churn        8      126
```

I decided that setting the cutoff threshold at 0.45 was optimal for my data mining task. The overall accuracy only decreased by roughly 2% when setting this threshold; moreover, and most importantly, the ***True Positive Rate*** increased by almost 6%, all the way up to 94%. In fact, the model only had issue correctly inferring 8 of the total 134 churn customers in the validation data.

Cumulative Gains Chart and Lift Chart Using New Prediction Threshold

Updating my working 'predicted classes' data frame:

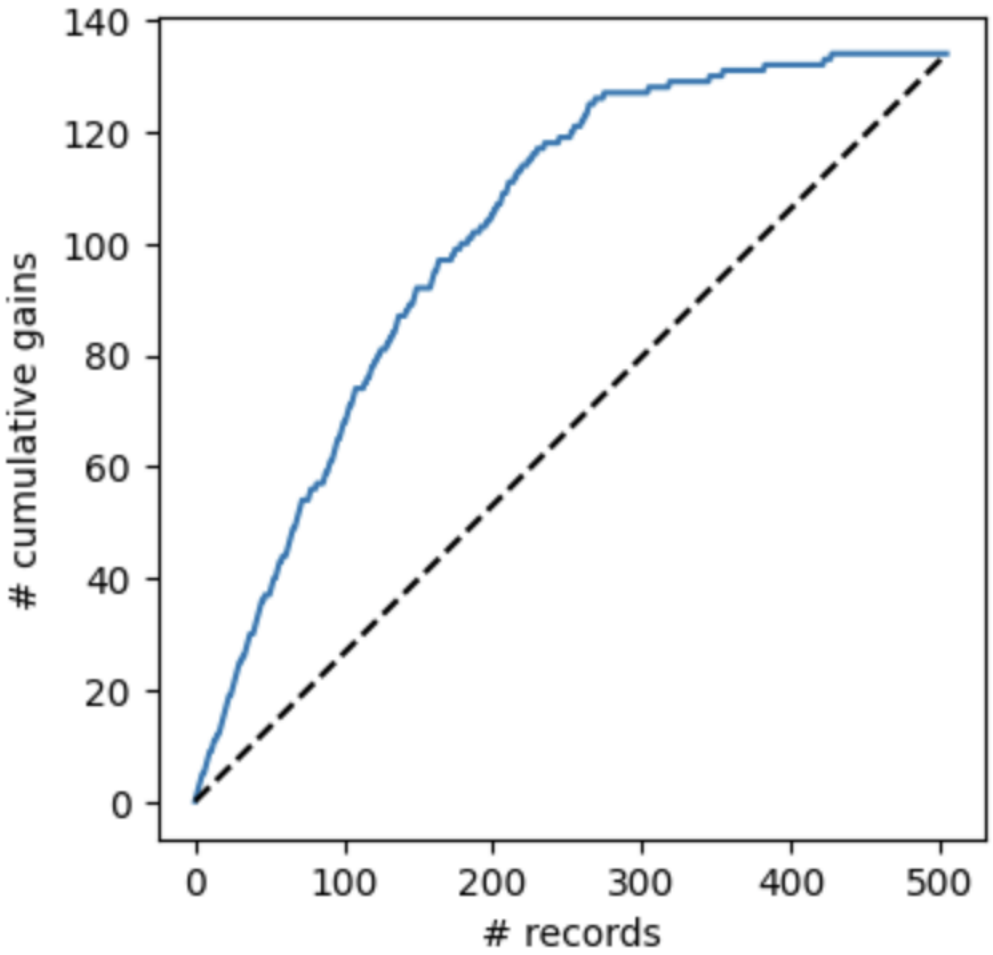
```
df['cumulative_actual_class'] = np.arange(1, 506)
```

```
df = df.assign(
    predicted = [1 if p > 0.45 else 0 for p in df.prob_churn]
)
```

```
df.head()
```

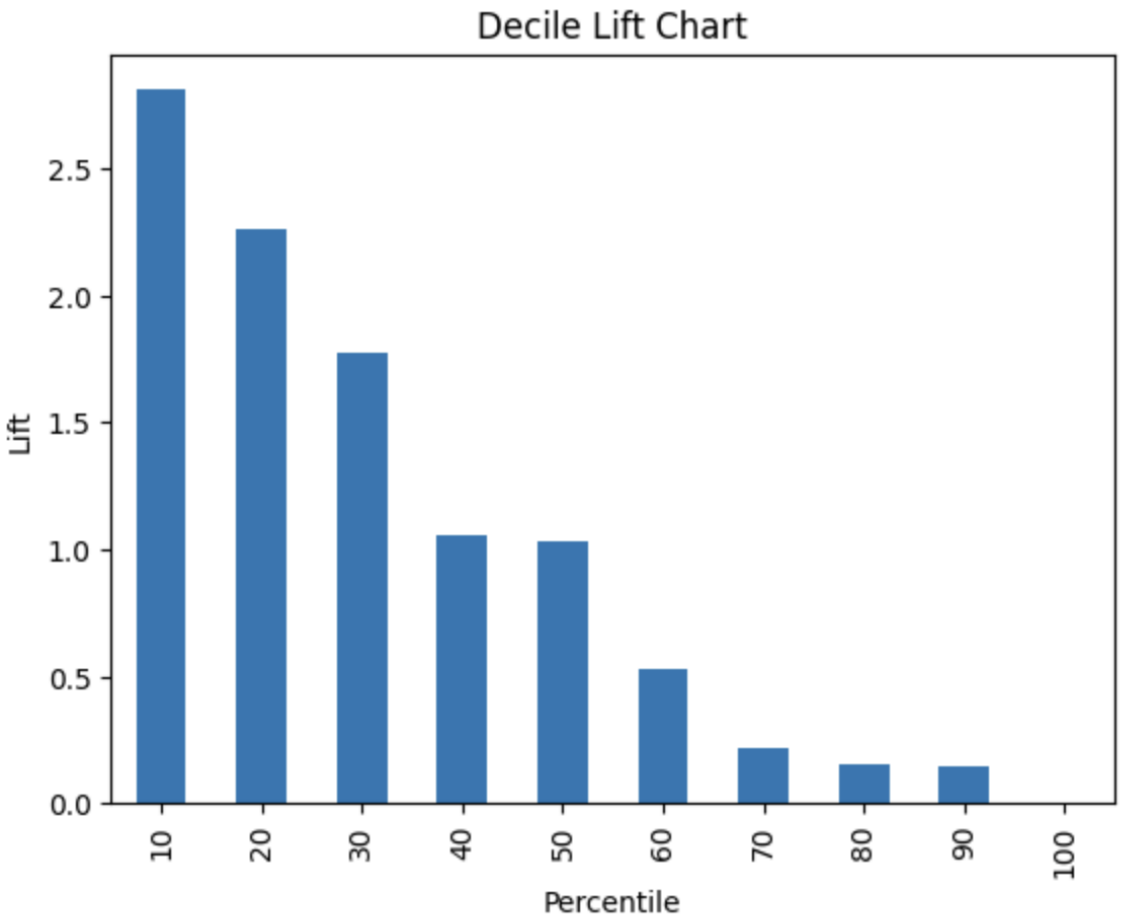
customerID	predicted	actual	prob_churn	cumulative_actual_class
7024-OHCCK	1	1	0.954586	1
4695-WJZUE	1	1	0.942162	2
8884-ADFVN	1	1	0.941698	3
2446-BEGGB	1	1	0.941602	4
1724-BQUHA	1	1	0.940507	5

Cumulative Gains Chart



Interpretation: From the validation data, consisting of 502 customers, 140 of them churned from TelCo. If we were to select 100 customers at random from the data, about 20-25 of them would belong to the 'churn' class. By using our model's predictions and sorting the customers by most to least likely to churn (based on the model's assigned propensities, at a cutoff threshold of 0.45), if we were to select the 100 customers predicted to be most likely to churn from TelCo, we would be right for about 60 of them.

Decile Lift Chart



The top 10% of customers that are ranked as 'most likely to churn' by our model yields over 2.5 more times as many actual churn customers compared to a random selection of 10% of customers from the validation data.

Given the insights provided by the above graph, I decided it would be worth analyzing more closely the top 30% of the customers in the validation data and take note of what specific characteristics they share and what is causing these 'churners' to leave TelCo, to see if the findings were agreeable with my earlier discoveries.

As such, I will make use of a 'Decision Tree' for visualization purposes, fitted to the top 30% of customers most likely to churn. This information will prove valuable, as I will be able to directly discern which characteristics interact in what manner to determine churn.

Decision Tree for Closer Analysis of Top 30% of customers (Sorted by Churn Rate) in Validation Data

Below is how I went about carrying out this procedure, by merging the original validation data frame (consisting of 505 customers) and the obtained 'predicted classes' data frame:

```
merged_df = original_df.join(df.drop(['actual','cumulative_actual_class'], axis=1), how='inner')
```

```
merged_df.sort_values(by='prob_churn', ascending=False)
```

```
for_tree = merged_df.iloc[:152] # 30% of the top data
```

```
for_tree.churn.value_counts()
```

churn	count
0	106
1	46

```
X = for_tree.drop(['partner', 'predicted', 'prob_churn', 'churn'], axis=1)
```

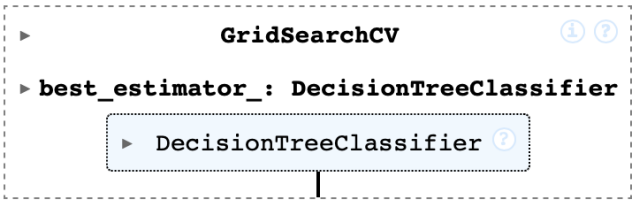
```
y = for_tree.churn
```

```
dt = DecisionTreeClassifier(random_state=0,)
```

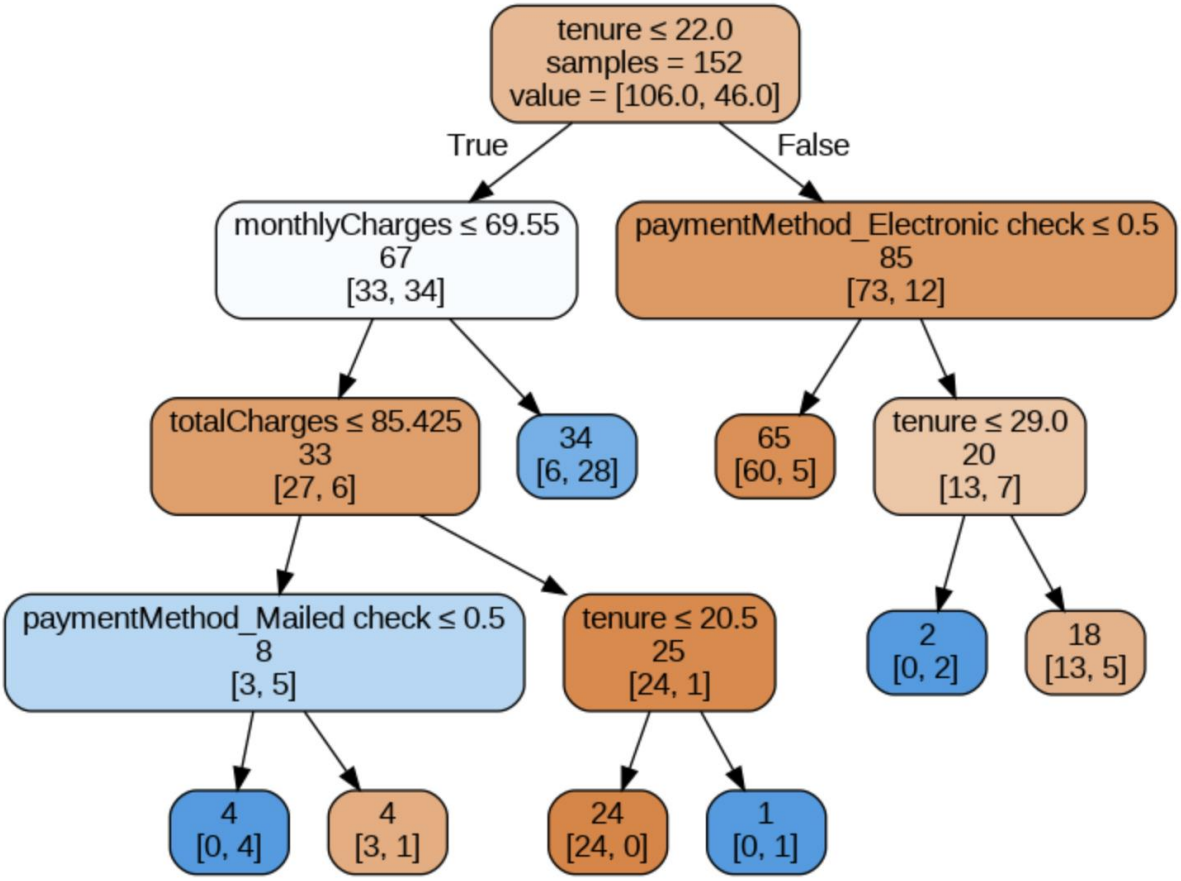
```
param_grid = {
    'max_depth': list(range(1,13,2)),
    'min_samples_split': list(range(2, 17, 2)),
    'min_impurity_decrease': [0, 0.001, 0.005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5]
}
```

```
dt_grid = GridSearchCV(
    dt, param_grid, n_jobs=-1
)
```

```
dt_grid.fit(X, y)
```



Resulting Tree:



We see that the customers from this dataset who have been with TelCo for less than or equal to 22 months *and* have monthly charges *greater than* $69.55 are more likely to leave. In fact, slightly more than 82% of the 34 customers belonging to this group did in fact churn from TelCo.

However, it is noticeable that, on the validation data, the Tree did *not* use the "Fiber Optic" variable as a split. However, based on the earlier insight, I decided to analyze this specific node further and see how many of these customers had purchased the "Fiber Optic Service" from TelCo.

As identified from an earlier Decision Tree, it is worth analyzing the customers belonging to this group and seeing how many of them have or had the Fiber Optic internet service package from TelCo.

Below details how I went about solidifying my hypothesis:

```
merged_df.columns = [col.replace(' ','_') for col in merged_df.columns]
```

```
merged_df.iloc[:152].query("tenure <=22 & monthlyCharges>69.55 & predicted==1 & internetService_Fiber_optic==1")
```

```
].query("tenure <=22 & monthlyCharges>69.55 & predicted==1 & internetService_Fiber_optic==1").churn.value_counts()
```

churn	count
1	27
0	5

dtype: int64

Business Analysis: We see that 32 of the 34 customers that belonged to that group *did in fact* have the Fiber Optic internet service from TelCo. We also see that, of those 32 customers, only 15% of them did not churn.

Recommendation & Strategy: The Fiber Optic service is not meeting customer expectations. I believe that TelCo is seeing a good number of new customers switching *to* TelCo and purchasing Fiber Optic internet service; however, the monthly rates they are seeing by subscribing to this service is not worth the perceived value that they are placing on its performance. The combination of high monthly charges and low value provided by the Fiber Optic internet package is a *primary* and undeniable factor leading to the high churn rate that TelCo is experiencing.

To prevent more churn, it is vital that TelCo lowers the monthly cost of this service and, while the cost is brought down, focuses on bringing the performance of the Fiber Optic service up to at least the performance and speed provided by its competitors. **Upon correcting the Fiber Optic issue, TelCo would see up to a 17% decrease in churn rate.**

Final Notes

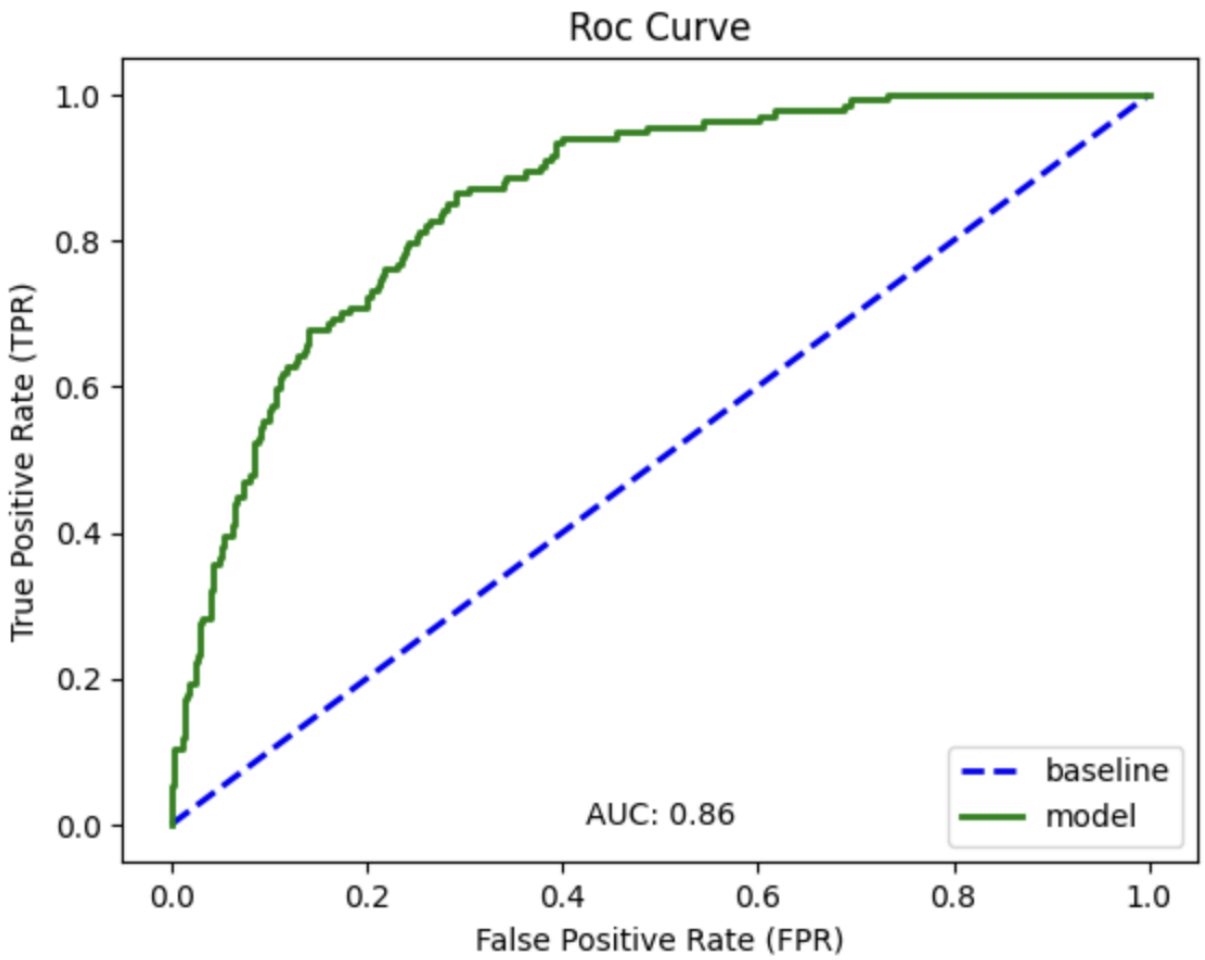
- ***See 'Appendix F' for K-Nearest Neighbors implementation.***
- ***See 'Appendix G' for results on unseen Test Data***

Appendix A

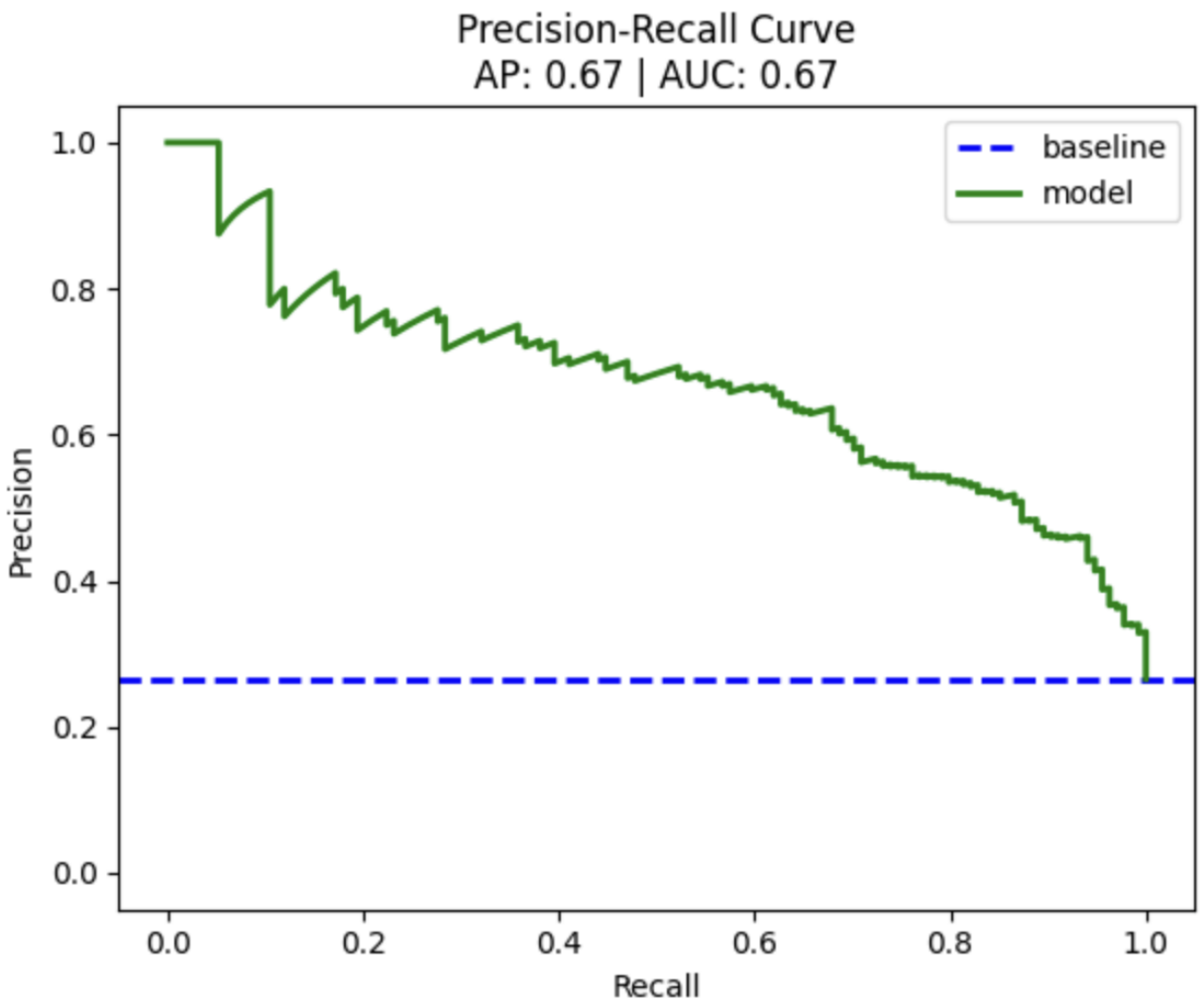
Classification Report for Baseline Model

	precision	recall	f1-score	support
0	0.92	0.75	0.82	371
1	0.54	0.81	0.65	134
accuracy			0.76	505
macro avg	0.73	0.78	0.74	505
weighted avg	0.82	0.76	0.78	505

ROC Curve for Baseline Model



Precision-Recall Curve for Baseline Model



Appendix B

Classification Report for 'Hyperparameter Tuning of Baseline Model (Attempt #1)'

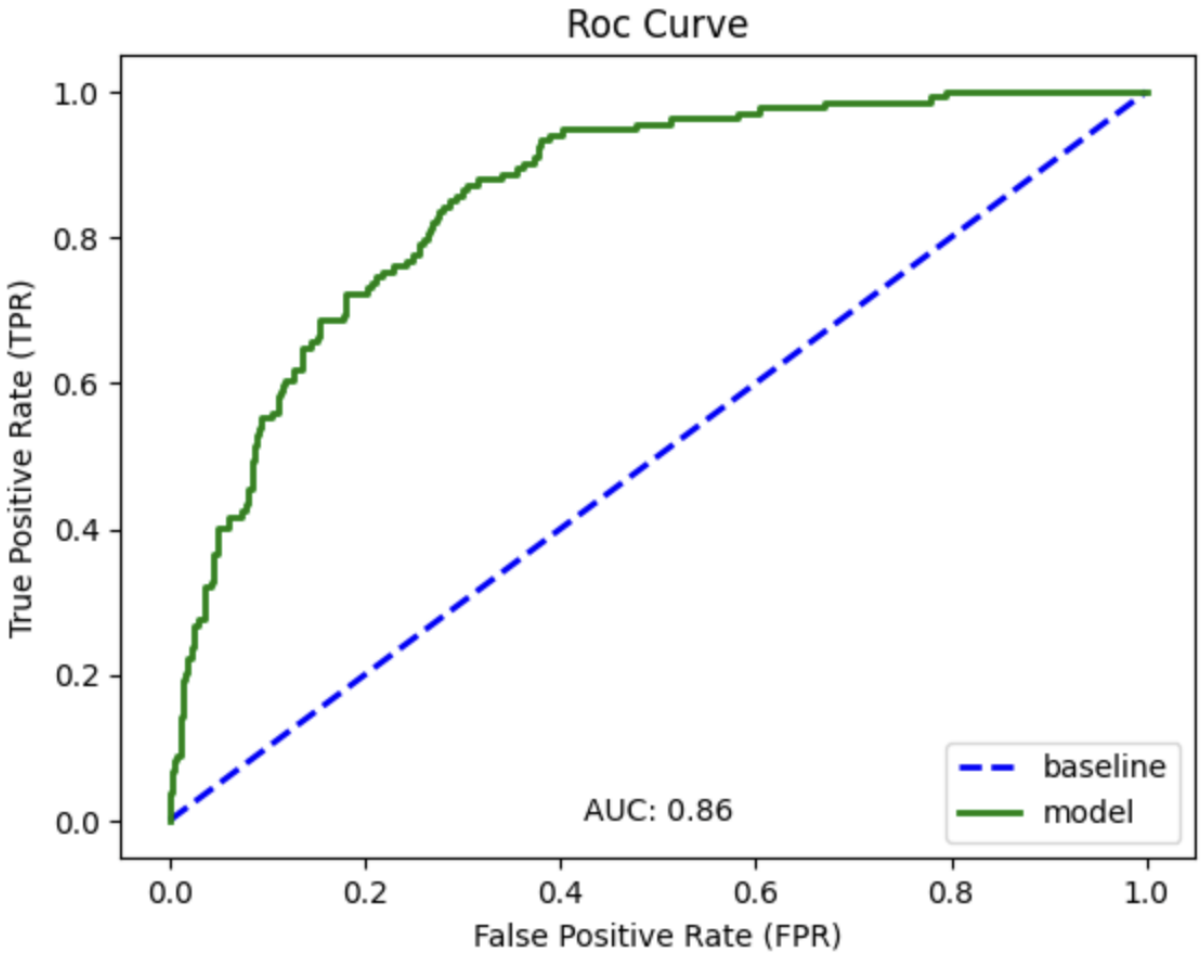
	precision	recall	f1-score	support
0	0.92	0.74	0.82	371
1	0.53	0.83	0.65	134
accuracy			0.76	505
macro avg	0.73	0.78	0.73	505
weighted avg	0.82	0.76	0.77	505

Appendix C

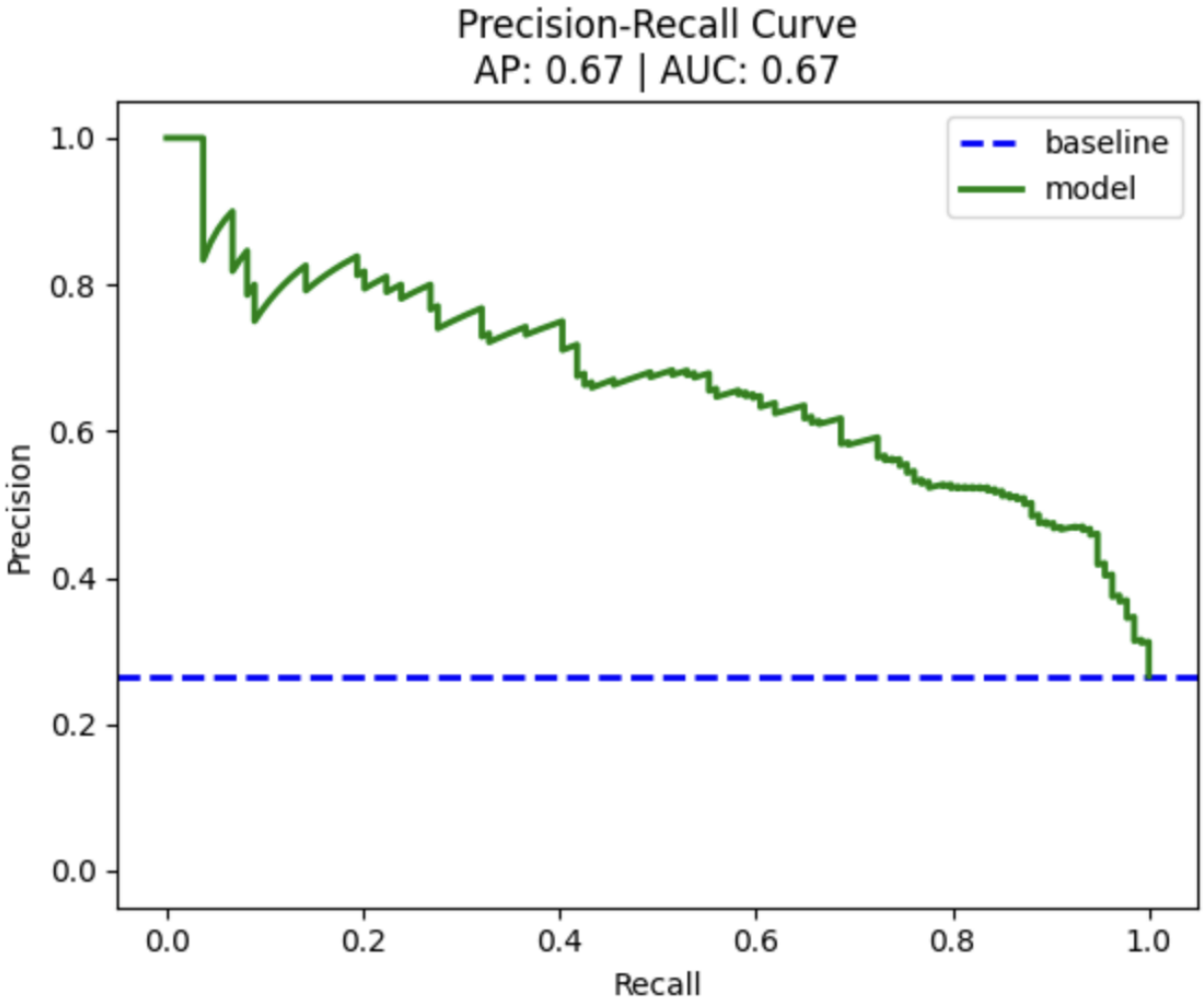
Classification Report for 'Hyperparameter Tuning of Baseline Model (Attempt #2)'

	precision	recall	f1-score	support
0	0.94	0.65	0.77	371
1	0.48	0.89	0.62	134
accuracy			0.71	505
macro avg	0.71	0.77	0.70	505
weighted avg	0.82	0.71	0.73	505

ROC Curve for 'Hyperparameter Tuning of Baseline Model (Attempt #2)'



ROC Curve for 'Hyperparameter Tuning of Baseline Model (Attempt #2)'



Appendix D

Classification Report for Fine-Tuned C Hyperparameter of Logistic Regression Model

	precision	recall	f1-score	support
0	0.94	0.66	0.77	371
1	0.48	0.89	0.63	134
accuracy			0.72	505
macro avg	0.71	0.77	0.70	505
weighted avg	0.82	0.72	0.74	505

Appendix E

Confusion Matrix (Accuracy 0.7604)

Actual	Prediction no churn	churn
no churn	277	94
churn	27	107

Appendix F (K-Nearest Neighbors)

Why K-Nearest Neighbors? Given the data mining task of predicting whether a customer would churn or not churn given their features, this classification technique made sense to implement. In **KNN**, records are classified as either 'churn' or 'no churn' based on the response values of their 'K nearest neighbors'. *Nearest*, in this sense, to a record implies that the features of the nearby data points are more similar to that particular record that any other point in the dataset. The algorithm, specifically, measures 'nearest' based on the Euclidean distance between the records; therefore, it is important that the data be scaled so the results aren't skewed.

Implementation

NOTE: *Here, I will simply outline the steps I followed in implementing this technique into my data mining pipeline. Please see 'Semester Project_4.ipynb' for more.*

1) **Split the data into Training and Validation subsets, using the same random seed used in all my prior models.**
2) **Standardized the Continuous features of the training data using the StandardScaler() class from scikit-learn.**
3) **Transformed the continuous features of the validation data into Z-scores using the mean and standard deviations obtained from standardizing the training data.**

4) **Determined the best value of K by analyzing the accuracy rate on the validation for a range of K values. The best value of K determined was K=7 to maximize accuracy. However, I needed to adjust the scoring metric because I aimed to choose K based on maximum TPR and minimized FNR.**
5) **The best K to optimize TPR on the validation data was K=5; however, the TPR for K=5 did not come close to comparing with the best TPR classification accuracy obtained in my fine-tuned Logistic Regression model. Therefore, I did not go forward in scrutinizing this technique any further.**

Appendix G (Test Data)

Why 'Test Data'? This data acts as a proxy for real-world data, to estimate how well the model will perform upon deployment in a practical operational setting. Specifically, in the given business context, as TelCo attracts new customers, they will be able to perpetually infer the probability that a customer will remain loyal customers or defect to the competition. This serves as a tremendous competitive advantage for TelCo, as they can monitor any given customers likelihood of churning, and, if necessary, take appropriate actions to keep these probabilities low across their customer base.

Implementation

NOTE: *Here, I will simply outline how I obtained this data and the results. Please see 'Semester Project_5.ipynb' for more.*

Implementation

This additional data was obtained from Kaggle. Upon importing the data into my python environment, I discovered that it contained 7043 customer records and 20 features. Moreover, the feature variables were identical to that of the 'TelCo Customer Data' that I had imported and used in training and validating my model. I removed the duplicate customer rows (that had already been used in building my model), and that left me with 2002 additional customer records to ensure the validity of my model.

Results with a Cutoff Threshold of 0.5

```
Confusion Matrix (Accuracy 0.6848)

          Prediction
  Actual no churn    churn
no churn      907      562
   churn       69      464
```

- **Accuracy**: 69%
- **Misclassification Rate**: 31%
- **True Positive Rate**: 87%
- **False Negative Rate**: 13%

Results with a Cutoff Threshold of 0.45

```
Confusion Matrix (Accuracy 0.6533)

          Prediction
  Actual no churn    churn
no churn      822      647
   churn       47      486
```

Total Customers in Test Data: 2002

- **Accuracy Rate**: 65.33%
- **Misclassification Rate**: 34.67%
- **True Positive Rate**: 91.18%
- **False Negative Rate**: 8.82%