Machine Learning Engineer Capstone Project

I. Defining the Project

Overview:

Direct-to-consumer marketing is important to brands and is very valuable. It allows companies to build a relationship with customers that fosters greater retention and loyalty. Leveraging in-app messaging has several benefits. For one, it's generally cheaper to retain existing customers than it is to acquire new customers. Mobile app adoption and in-app commerce enables companies to leverage data to better understand user preferences. This creates an opportunity (and challenge) to personalize offers and marketing content for a given customer. All customers have different preferences and tastes. They also have different price elasticities and spending habits. Understanding which customers will be most receptive to a given offer represents a nuanced, yet important business challenge.

In the case of Starbucks, rewards members can transact via the app, as well as receive promotional offers. In this problem, I'll be using simulated data from the Starbucks rewards mobile app. The dataset includes transactional data showing user purchases made on the app including the timestamp of purchase and the amount of money spent on a purchase. This transactional data also has a record for each offer that a user receives as well as a record for when a user actually views the offer. There are also records for when a user completes an offer. In addition to the transaction data, we also have demographic data about the users (i.e. age, gender, income, etc.) and metadata on the offer (i.e. offer type, validity period, marketing channel, etc.).

Problem Statement:

Given that we know different users will respond differently to different offers, our task is to determine how likely a given user that receives an offer will convert based on information about the user and the offer itself. By 'convert', we mean the user takes advantage of the offer by meeting the requirements of the transaction. In practice, this model could be used to determine which offers should be given to particular users based on their likelihood of converting . We can also evaluate the which offers are users overall my responsive to, and what drivers (i.e. user demographics and offer metadata) are most correlated with conversion. This type of key driver analysis can offer some important insights to marketing teams on how to both tailor offers and target customers for future promotions.

Evaluation Metrics:

Our goal is to develop a binary classification model that best predicts how a user will respond to a given offer. The exact metric we'll use to measure the performance of this model will depend on a few different factors. The first consideration is whether or not we have a preference for reducing false positives or false negatives. Put another way, what is the potential cost of misclassifying a user likely to convert vs misclassifying a user not likely to convert. In this context, sending someone a marketing offer when they probably won't convert (or would have transacted without the offer) is a waste of the marketing budget. On the other hand, failing to deliver an offer that makes sense for the user is a lost opportunity. Without additional data on marketing costs or the incrementality associated with giving the offer, we'll make an assumption that both goals are important. In this scenario the f1 score which is the harmonic mean of precision and recall measures is a suitable evaluation metric. Another consideration is the mix of positive (conversions) and negative (non-conversions) observations in the training data. The assumption is that conversions we'll be less prominent than non-conversion events, but that the data will not exhibit extreme class imbalance (i.e. 98% non-conversions vs 2% conversion cases). We'll confirm this when conducting exploratory analysis of the dataset.

II. Analysis

Data Overview:

The data is contained in three files:

- portfolio.json - containing offer ids and meta data about each offer (duration, type, etc.)
- profile.json - demographic data for each customer
- transcript.json - records for transactions, offers received, offers viewed, and offers completed

Portfolio.json:

- id (string) - offer id
- offer_type (string) - type of offer i.e. BOGO, discount, informational
- difficulty (int) - minimum required spend to complete an offer
- reward (int) - reward given for completing an offer
- duration (int) - time for offer to be open, in days
- channels (list of strings)

Profile.json:

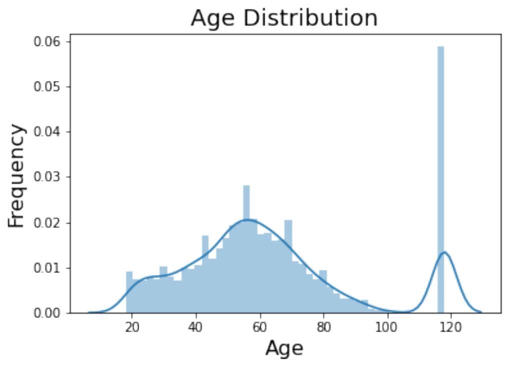
- age (int) - age of the customer
- became_member_on (int) - date when customer created an app account
- gender (str) - gender of the customer (note some entries contain 'O' for other rather than M or F)
- id (str) - customer id
- income (float) - customer's income

transcript.json:

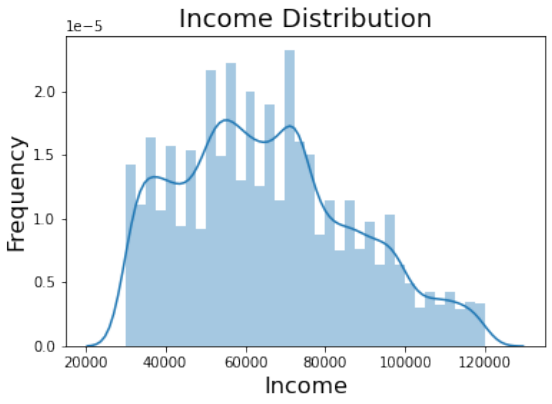
- event (str) - record description (i.e. transaction, offer received, offer viewed, etc.)
- person (str) - customer id
- time (int) - time in hours since start of test. The data begins at time t=0
- value - (dict of strings) - either an offer id or transaction amount depending on the record

Data Exploration and Preparation:

The profile data includes 17,000 records, and all the records in the 'id' field are unique. We can see that 'gender' and 'income' each have 2175 null records. We'll need to account for this if we use any of these fields in our final model. Looking at the age field we see the distribution appears relatively gaussian, though we see a spike in observations where the user age is 118 which is odd. We actually have 2175 records that have 118, which would suggest this may be a placeholder value for users where the age is not known.

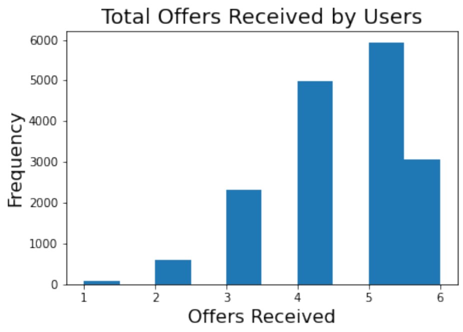


The income distribution is slightly right skewed which is not uncommon for many datasets with population income.

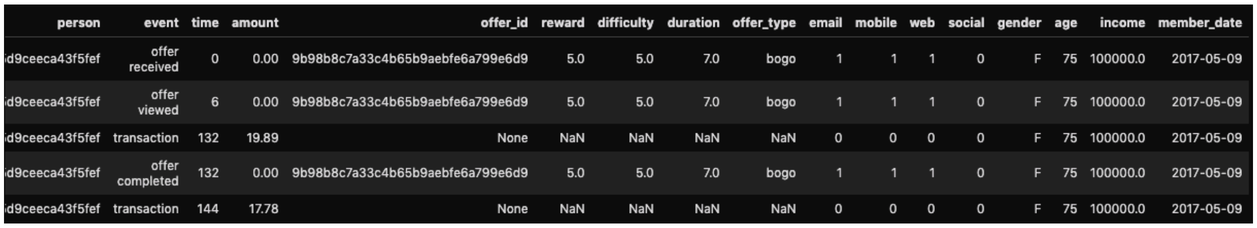


The portfolio df contains metadata for the suite of offers within the dataset. It includes information like the duration of the offer until it expires, the financial reward, and the relative difficulty of completing the offer. It also includes the 'channels' field which appears as a comma separated list. We'll need to parse these values into one-hot encoded fields.
The transcript df is an interesting dataset that includes an 'event' field which provides a timestamp when an offer is received, viewed, and completed by each user. It also captures each user's transaction. Transactions make up 45% of the records and offer completed events make up 11% of records. One important thing to note is that a user can have an offer completed event (i.e. the user satisfies the terms of the offer) without having an offer viewed event. In other words, the user can satisfy the terms without being necessarily influenced by the marketing promotion. We may choose to only label a conversion as a true conversion if the

user actually viewed the offer. The ‘value’ field will also need to be parsed. For events like ‘offer received’ and ‘offer completed’ the field contains the offer id and for transactions it captured the amount spent.



Combining the data frames:
We’ll simplify the analysis by creating a dataset that merges all 3 data frames, applying the data cleaning steps identified previously. After merging the data frames, we’ll need to parse the offer id and transaction amount from ‘value’ field in the transaction df, as well as the ‘channels’ field in the portfolio df. We’ll also convert the user’s ‘became_member_on’ to a formatted date.

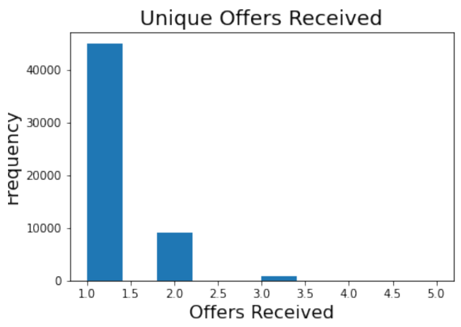


person	event	time	amount	offer_id	reward	difficulty	duration	offer_type	email	mobile	web	social	gender	age	income	member_date
d9ceeca43f5fef	offer received	0	0.00	9b98b8c7a33c4b65b9aebfe6a799e6d9	5.0	5.0	7.0	bogo	1	1	1	0	F	75	100000.0	2017-05-09
d9ceeca43f5fef	offer viewed	6	0.00	9b98b8c7a33c4b65b9aebfe6a799e6d9	5.0	5.0	7.0	bogo	1	1	1	0	F	75	100000.0	2017-05-09
d9ceeca43f5fef	transaction	132	19.89	None	NaN	NaN	NaN	NaN	0	0	0	0	F	75	100000.0	2017-05-09
d9ceeca43f5fef	offer completed	132	0.00	9b98b8c7a33c4b65b9aebfe6a799e6d9	5.0	5.0	7.0	bogo	1	1	1	0	F	75	100000.0	2017-05-09
d9ceeca43f5fef	transaction	144	17.78	None	NaN	NaN	NaN	NaN	0	0	0	0	F	75	100000.0	2017-05-09

Creating the Conversion Table:
The goal is to create a table that has all of the 'offer received' events for every user. This table will be the foundational data we'll use to model the likelihood that an offer will be completed. In addition to the time the offer was received, we'll also include the offer metadata (i.e. reward, duration, difficulty) and the user demographic data (i.e. income, gender, and customer tenure.

Additionally, we'll create binary flags for if the offer was viewed and completed by the user. The model response will be the offer completed flag.

Prior to creating this data source we'll need to validate some assumptions about the data. One assumption is that users received the same offer based on the offer id only once.



We can see that while most users received a given offer once, 15% of users received the same of 2+ times (with some receiving the same offer up to 5 times over the test period).
Given that users can potentially receive the same offer multiple times, we'll simplify the analysis by taking the first record of a given event for each user/offer combination. Meaning if a user received/viewed the same offer on separate instances, we'll exclude all but the first record.

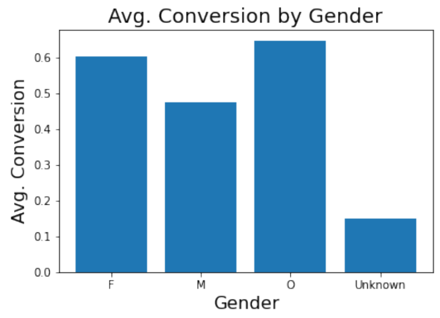
Creating the conversion table will require the following steps: 1) Create 3 temp tables that contain the first occurrence for all events (offer received, offer viewed, and offer completed), 2) merge the tables, 3) create binary flags for whether the offer was viewed and/or completed, and 4) create a binary flag that records a conversion if the user both viewed then completed the offer. The goal of our model to predict whether a given offer will impact whether the user completes it, so we'll define conversions by observations where the user was presumably influenced by the offer.

Exploratory Visualization:

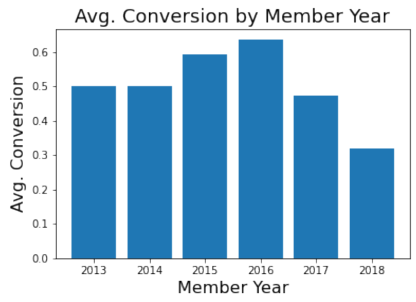
We'll now focus on analyzing the properties of the dataset prior to modeling. Specifically, we'll look at the relationship between the features and the response (offer conversion). This analysis will help provide some intuition about which features may be most/least useful for the modeling task. Also, we need to assess how the features are related to one another. Multicollinearity exists when 2 or more features are highly correlated with one another. We seek to avoid this for several reasons. Firstly, we should seek to make our model as parsimonious as possible, as simpler models make it easier to explain what is driving the prediction. If two features are highly correlated, then using just one should be enough to capture the relationship with the response. Also, the existence of multicollinearity can make interpreting the key drivers of the prediction unreliable when using methods like feature importance and SHAP plots. In practice, business stakeholders at Starbucks will most certainly be interested in understanding these relationships as well.

Firstly we can see that 39% of all sent offers (63K total) were converted. When we examine the conversion rate by offer type, we see that discount offers have a slightly higher conversion rate than bogo offers (50% vs 47%). More interesting is that informational offers have a 0% conversion rate which suggest that we these offers technically cannot be completed. In this case these offers really should not be in our dataset. Informational offers represent 20% of all records and removing them would reduce our dataset to about 50K which while not ideal, should still be enough data to model the problem. After removing informational offers, discount and bogo offers now each have 50% share of all offers. The new response distribution is 48% conversions vs 52% non-conversions.

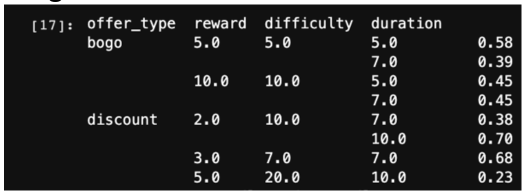
The offer viewed flag is likely very useful in predicting conversion and in fact, users that viewed the offer converted 61% of the time. Starting with the categorical features, there appears to be some variability in conversion by gender, with women converting at 60% and men at just 48%. Unknown users (users with null demographic data) have the lowest conversion at 15%. This would indicate that this this subset of users is not representative of the entire population.



The relationship between the year the member created their account and the avg conversion is a bit inconsistent. Newest customers clearly display a lowest conversion. But the relationship for the remaining years is not consistent.



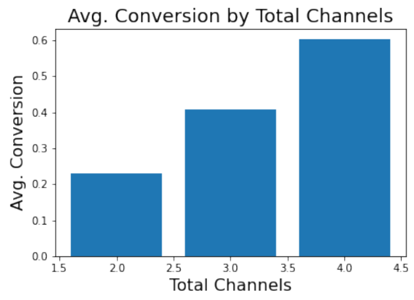
A simple analysis of the conversion of individual offer types can provide some very useful insight for marketers.



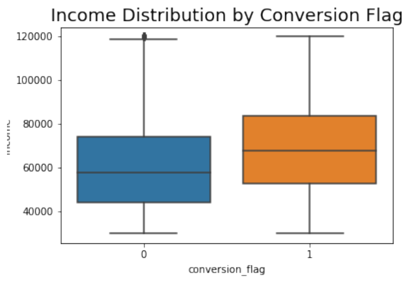
```
[17]: offer_type  reward  difficulty  duration
      bogo        5.0     5.0         5.0         0.58
                                      7.0         0.39
                  10.0    10.0        5.0         0.45
                                      7.0         0.45
      discount    2.0     10.0        7.0         0.38
                                      10.0        0.70
                  3.0     7.0         7.0         0.68
                  5.0     20.0        10.0        0.23
```

The discount offer with the 2 dollar reward and 10 day duration has significantly higher conversion vs the same offer with the 7 day duration (70% vs 38%). The same trend doesn't hold true for the BOGO offers, however. The 10 dollar BOGO offer has the same avg. conversion regardless of duration. In fact, the 5 dollar bogo offer with the shorter 5 day duration has 58% avg. conversion vs 39% for the offer with 7 day duration.

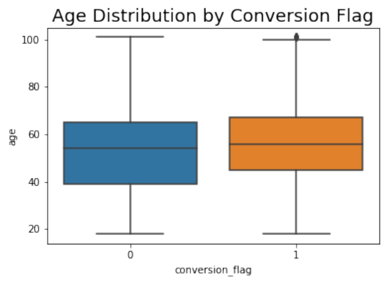
There appears is a positive relationship between the number of channels used to distribute the offer and avg. conversion. To simplify the model features, we'll likely keep the total number of channels as opposed to the individual channel flags.



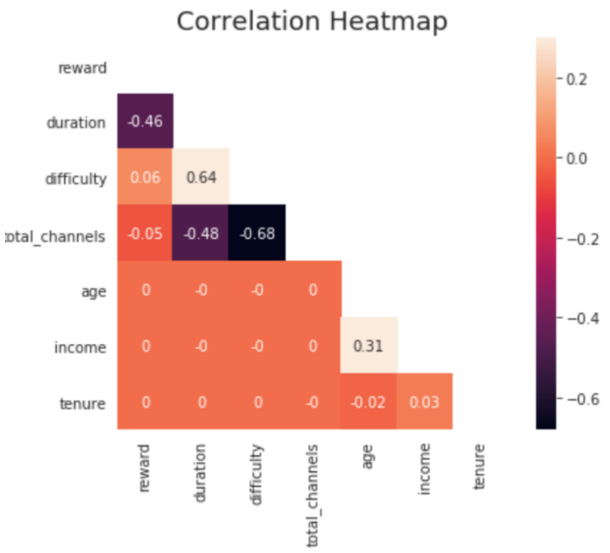
Next we'll look at the distribution of the numeric features and the response starting with income.



We do see that in instances where a conversion occurred, the income distribution of those users is slightly higher. When we do the same analysis with age, we see that the median age of both converters and non-converters is the same.



Next we'll look at the correlation among the numeric features to assess if multicollinearity exists using a correlation heatmap, which denotes the Pearson correlation coefficient.



There are no features with strong correlation between them. The total channels and offer difficulty have a negative correlation of -0.68, indicating that offers sent through more channels are likely to be less difficult to complete.

Algorithms and Techniques:

The first note is that we'll be using Amazon Sagemaker and its available built-in algorithms to predict conversion. Amazon SageMaker is a fully managed service that provides every developer and data scientist with the ability to prepare build, train, and deploy machine learning (ML) models quickly. SageMaker removes the heavy lifting from each step of the machine learning process to make it easier to develop high quality models. This will provide an efficient framework to quickly train and easily deploy our model to production systems in the future. Specifically, we'll be using the Linear Learner and XGBoost algorithms for this problem. Both support binary classification problems and allow us to utilize our chose evaluation metric, the f1 score. In the case of Linear Learner, the algorithm learns a linear threshold function, and maps a vector x to an approximation of the label y. The algorithm supports three data channels: train, validation (optional), and test (optional). The algorithm logs validation loss at every epoch, and uses a sample of the validation data to calibrate and select the best model. For training, the linear learner algorithm supports both recordIO-wrapped protobuf and CSV formats. For inference, the linear learner algorithm supports the application/json, application/x-recordio-protobuf, and text/csv formats. One reason we'll leverage this algorithm is because it provides the flexibility to train on single- or multi-machine CPU and GPU instances, providing flexibility to handle larger datasets efficiently. Another advantage is that it allows for streamlined data preprocessing with automatic feature scaling. The algorithm trains with a distributed implementation of stochastic gradient descent (SGD), though the optimization algorithm can be changed.

We'll also use the Sagemaker implementation of the XGBoost algorithm, a popular and efficient open-source implementation of the gradient boosted trees algorithm. Some of the benefits of the algorithm is its performance due to providing an ensemble of predictions and it's robust handling of a variety of data types, relationships, and distributions. It doesn't require some of the preprocessing necessary for linear models such as feature scaling and handling outliers. In Sagemaker, you can use XGBoost as a framework (similar to how one would use PyTorch or MXNet), with custom training script, or simply use the Sagemaker's built-in implementation of the algorithm. The SageMaker implementation of XGBoost supports CSV and libsvm formats for training and inference. SageMaker XGBoost uses the Python pickle module to serialize/deserialize the model, which can be used for saving/loading the model. SageMaker XGBoost 1.0-1 or earlier currently only trains using CPUs. It is a memory-bound (as opposed to compute-bound) algorithm. So, a general-purpose compute instance (for example, M5) is a better choice than a compute-optimized instance (for example, C4).

Sources:
https://aws.amazon.com/sagemaker/features/
https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner.html
https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html

Benchmark:

The first to keep in mind in terms of benchmarking the performance of the conversion prediction model is the proportion of the dominant class of the response (conversion flag). In our case, we see that 48% of offers in the full dataset converted, while 52% did not convert. The 52% is an important benchmark because it means that we could presumably achieve 52% accuracy if we just classified each observation as a non-conversion. Of course this would not be a useful model, but any model we create would at the bare minimum need to provide higher accuracy than 52%. This is also known as the naïve model, or naïve solution. Our goal will be to improve upon this benchmark during the modeling process.

III. Methodology

Data Preprocessing:

The conversion table dataset (df_conversion.pkl) requires several preprocessing steps to prepare the data for modeling. We first need to remove observations where the offer type is informational. During EDA we discovered that these offers are technically not recorded as completed because they don't require the user to transact to satisfy the offer. Next we need to remove unneeded features such as user ids, offer ids, and other fields we created for analysis purposes only. Also, we observed that a large number of users had the age of 118, which we can assume is bad data. We'll convert these values to nulls so that we can impute them later. Next we split the data into the training, validation, and test sets. The training and validation datasets will be used by Sagemaker to train the algorithms and we'll evaluate the performance of the model on the unseen test set. To accomplish the data transformation steps we'll utilize Scikit Learn Pipeline. Pipeline will allow us to apply a sequence of data transformation steps such as scaling and imputing nulls to each dataset (i.e. training, validation, and test). I should note the importance of splitting the data prior to applying these transformations, as this can bias the model performance results. For example, if we were to impute the missing income values across the entire dataset prior to splitting the data we would unknowingly be introducing information about the test data into the training dataset. This is known as data leakage. We fit the pipeline steps to the training data and use it to transform not only the training data but also the validation and test datasets. Because we intend to use a linear model (Sagemaker Linear Learner) we will scale the numeric features so that they all have a mean of 0 and standard deviation of 1. Linear models are sensitive to features with vastly different scales. In our case we have age which has values less than 100 and income which has values in the hundreds of thousands. Additionally, we impute the null values for income and age. Both features had distributions that appeared relatively Gaussian so we'll impute them with the median. In this case mean imputation would also be appropriate as those values would be similar based on the distribution. The XGBoost model is not affected by the scale of the data, but can handle both scale and unscaled data. We'll also one-hot encode the categorical features such as offer type and gender. One should be cautious as one-hot encoding with many categorical features as it can increase the dimensionality of the dataset quickly which can negatively impact the performance of some models. After applying these preprocessing steps to the data we'll save the data both locally and upload to S3 so that it can be accessed by Sagemaker.

Implementation:

The process for which metrics, algorithms, and techniques were implemented with the given datasets or input data has been thoroughly documented. Complications that occurred during the coding process are discussed. Walk through the steps of the coding notebook – model 1. The first algorithm we'll use to predict conversion is the Linear Learner algorithm available via Sagemaker. We first import standard libraries for data manipulation such as NumPy and pandas. We'll also import the Sagemaker Python SDK and the AWS Python SDK (Boto3). We then set the 'session' variable. This is an object that represents the SageMaker session that we are currently operating in and contains some useful information that we will need to access later such as our region. Then we set the 'role' object. This is an object that represents the IAM role that we are currently assigned. When we construct and launch the training job later, we will need to tell it what IAM role it should have. Next, we'll specify the location of our data in S3 which will be referenced by the training job. I should note that we're also specifying the location of the test data as we'll use this to evaluate the performance of the classifier. We assign the Sagemaker training input objects for both the training and validation data. Now it's time to execute the training job. To accomplish this we first use the sagemaker module to retrieve the Docker image that contains the Linear Learner algorithm. We then give the training job a unique name and specify the S3 bucket location where we want to store the trained model. We then create the linear learner estimator object which specifies the container location, role, output path, etc. We selected instance type ml.m4.xlarge, as this was recommended in the AWS resources. For this initial model we will set default hyperparameters rather than letting Sagemaker learn the appropriate hyperparameter values via a Hyperparameter Tuning Job. A few to note are that we set the learning rate, which is the step size used by the optimizer for parameter updates, and the wd which is the weight decay parameter, also known as the L2 regularization parameter. If you don't want to use L2 regularization, set the value to 0. You can see that the algorithm will scale the features if normalize_data=True. We actually did this manually in our preprocessing notebook, but Linear Learner will conveniently do that task as well. Next we fit the model, utilizing the specified resources for training. In this example, we'll also deploy the model to a live endpoint. This would be useful if we want to make our model available to other systems in production in the future. Perhaps we'd want to make these predictions available to an email marketing system like Salesforce Marketing Cloud so that we could target users for different offer emails based on the user and the offer. Next we load the test data from S3 and generate predictions using our model. The prediction result includes both the predicted probabilities (i.e. the likelihood of conversion from 0-1) and the inferred classification derived by the model. The f1 score on the test data is 0.809 and the accuracy is 0.791 on the test data. The f1 score is maximized when the predicted probability threshold is set to 0.420513. This is certainly an improvement to the naïve model accuracy of 0.52 which could be achieved by simply predicted that all offers were not converted. Next we'll attempt to improve on our solution through additional models.

Refinement:

Our first attempt at improving on the current solution will be to allow Sagemaker to learn the optimal Linear Learner model hyperparameters via a Hyperparameter Tuning Job. After we create the estimator object, we then create the HyperparameterTuner object. We specify the estimator object, the objective metric (binary_f_beta), among other options like the max_jobs which is maximum total number of training jobs to start for the hyperparameter tuning job. By default, the process will learn hyperparameters through Bayesian optimization. We could also specify early stopping, which specifies whether early stopping is enabled for the job. Can be either 'Auto' or 'Off' (default: 'Off'). If set to 'Off', early stopping will not be attempted. If set to 'Auto', early stopping of some training jobs may happen, but is not guaranteed to. Some of the hyperparameters we will tune are the learning rate, wd/l1 which perform different kinds of regularization, as well as the mini batch size. AWS provides very useful documentation on which hyperparameters to tune and the range of values to search through (https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner-tuning.html). Also note that Sagemaker does not perform GridSearch, meaning that this process will not build a model for every combination of the hyperparameters. It uses Bayesian optimization to learn the best hyperparameter values faster and more efficiently. After running the job, Sagemaker conveniently allows us to create a new estimator object using the best model from the hyperparameter tuning job. We can deploy this model in the same fashion we did previously. We then import the test data from S3 and make predictions on it. Evaluating the performance, we get an f1 score of 0.804 (on par with but slightly lower than the original model f1 score of 0.809) and an accuracy of 0.787 (vs 0.791 in our first model). So the model performance on the test set slightly lower than our initial model. We save the predicted probabilities to S3 so we can use them for further analysis.

Finally, we'll attempt to improve on our initial solution by using the XGboost algorithm. In this example, we'll set up our training job a bit differently by using the CreateTrainingJob API which can also be used to start a model training job. After training completes, Amazon SageMaker saves the resulting model artifacts to an Amazon S3 location that you specify. If you choose to

host your model using Amazon SageMaker hosting services, you can use the resulting model artifacts as part of the model. You can also use the artifacts in a machine learning service other than Amazon SageMaker, provided that you know how to use them for inference. The training job parameters that need to be specified are mostly the same. We need specify the location of the model container, input data, IAM, role, instance type, etc. We also need to set the hyperparameter values. A few we'll specify include the max depth, which is the maximum depth of a tree. Increasing this value makes the model more complex and likely to be overfit. The subsample is the ratio of the training instance. Setting it to 0.5 means that XGBoost randomly collects half of the data instances to grow trees. This prevents overfitting.
The eta represents the step size shrinkage used in updates to prevent overfitting. After each boosting step, you can directly get the weights of new features. The eta parameter actually shrinks the feature weights to make the boosting process more conservative. We'll set these values using some recommended values by AWS. After setting the training parameters, we then use the Sagemaker session object to create and execute the training. Rather than deploying our model to a live endpoint this time, we'll generate predictions using a Batch Transform job. This will allow us to allocate AWS resources to use our model to generate predictions on a single batch of input data. This is useful when we don't need to make predictions in real-time, as is the case for this one-time analysis. The prediction output of the Batch Transform job is saved in S3. We read the object from S3 and parse the predicted probabilities. We can convert the probabilities to classifications using a threshold of 0.5. Comparing this to the actuals, we see that the f1 score is 0.811, slightly higher than the initial Linear Learner model f1 score of 0.809. The accuracy of 0.804 is also slightly higher compared to 0.791. When we calculate the f1 score uses a range of predicted probability thresholds (as opposed to the default of 0.5) we see that the f1 score is maximized at 0.816 when the threshold is 0.398.

IV. Results

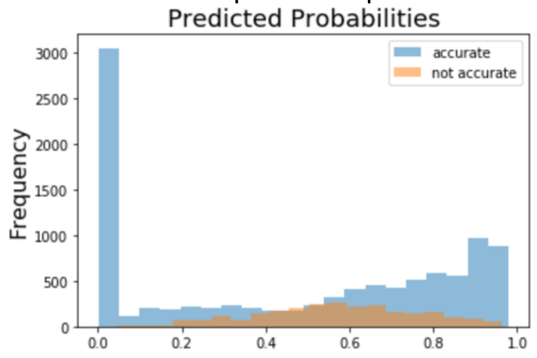
Model Evaluation and Validation:

To summarize our results, the XGBoost model resulted in the highest f1 score for modeling offer conversion on the test dataset at 0.816 (when the predicted probability threshold was set to 0.398). Our initial Linear Learner model resulted in an f1 score of 0.809 and the tuned Linear Learner model had an f1 score of 0.807. Diving into the metrics more deeply, we can see that the XGBoost model classifies 86.8% of actual conversions correctly (Sensitivity) and 74.4% of actual non-conversions correctly (Specificity). When compared to the initial Linear Learner model, the sensitivity is higher at 91.5%, but the specificity is significantly lower at just 67.4%. This means that the Linear Learner model is better at classifying conversions events, but worse at classifying non conversion events. This stresses the importance of understanding the business use case. One of the premises of our analysis is that we care equally about identifying conversions and non-conversion events. The XGBoost model in a sense provides a more balanced prediction, in which it does a better job at classifying non-conversions but the price is that it doesn't do quite as well at classifying conversions. We may have a use case where we're more interested in correctly classifying converters, in which case the Linear Learner model may be more appropriate. It's also possible tin some cases to calculate the cost of a false positive vs a false negative and to use that costs to evaluate the classifier. This requires communication with business stakeholders to ensure the model meets the business needs.

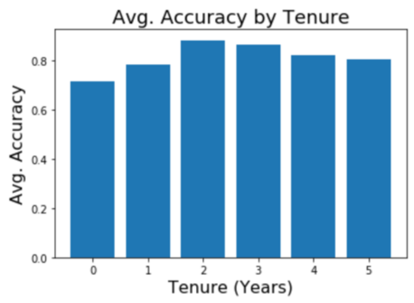
Post Hoc Analysis:

The final model which produces an accuracy of ~80% is a significant improvement to the naïve model of 52% (simply predicting all offers will not be converted). In the context of the problem, we would be using these predictions to decide whether to send or not send a user an offer based on their likelihood of converted. This level of accuracy could be very useful in this marketing context, and aside from the marketing costs and the potential for sending a user an offer for a purchase they would have made anyway, there isn't significant risk in using our model to target users.

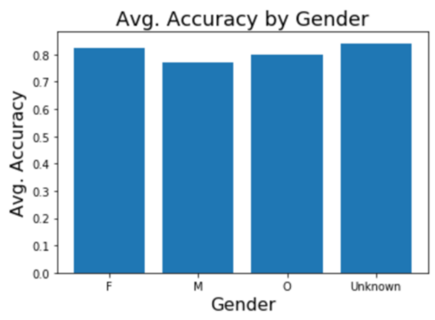
In addition to generating predictions, our final analysis task will be to assess the model's predictions accuracy based on the user demographics and the offer metadata. We'll need to import the predicted probabilities from our best model (XGBoost), the test data prior to scaling, and the test dataset response values. We'll append these to the test data frame and create calculated fields for the model prediction (using the threshold of 0.3987 which maximized the f1 score) and a binary flag for whether that prediction was accurate. Firstly, we can look at the distribution of the predicted probabilities and whether they were accurate or not.



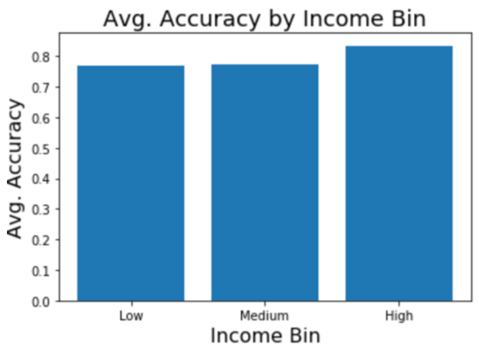
The plot shows the majority of misclassifications occurred when the predicted probability was in the middle range (0.4-07). There were fewer misclassification when the predicted probabilities were very small (.e. < 0.2). Conversely you can see that there are slightly more misclassifications on the high end of predicted probability. Next we can look at the accuracy broken out by various user demos.



We see that the accuracy is slightly lower for newer customers, those with less than 2 years tenure



The accuracy by gender is fairly consistent, though a bit lower for males.



Accuracy is slightly higher among high income users.

```
[26]: total_channels
      2.0    0.908575
      3.0    0.838453
      4.0    0.742172
      Name: accurate, dtype: float64
```

We can see pretty significant difference in the avg. accuracy by the number of channels used to communicate the offer. There is a negative relationship with the number of channels and the avg. accuracy.

```
offer_type  reward  difficulty  duration
bogo        5.0     5.0         5.0         0.713313
                                7.0         0.875866
            10.0    10.0        5.0         0.737978
                                7.0         0.769476
discount    2.0     10.0        7.0         0.870839
                                10.0        0.760994
            3.0     7.0         7.0         0.756975
            5.0     20.0        10.0        0.908575
```

The offer with the lowest conversion avg. accuracy was 5 dollar BOGO at 71.3%. The 5 dollar Discount had the highest avg. accuracy at 90.9%

V. Next Steps

There are several steps that can be taken to improve the model/analysis in future iterations. Firstly, it would be good to use traditional methods such as feature importance and SHAP values to understand which features are key drivers of the prediction. Though not out of the box, it is possible to accomplish this in Sagemaker. This type of insight would be extremely useful for helping marketers to craft successful offers based on these findings. Being able to surface these insights to users in a dashboard or app would also be a great addition.

Secondly , additional features can be introduced to the model such as customer transaction amounts. The current model is not taking into account the monetary value of the transactions the customer made during the test. Distinguishing users that completed a BOGO offer but spent $10 in the transaction compared to a user that spent $100 seems like it would be valuable information to introduce into the model. Also, calculating the number of times an offer was received/viewed prior to being completed could also be helpful.

Another helpful exercise would be to determine how long an offer should last. We could analyze what happens to the offer completing rates when testing different offer duration periods.