Direct Marketing with Amazon SageMaker XGBoost and Hyperparameter Tuning

Supervised Learning with Gradient Boosted Trees: A Binary Prediction Problem With Unbalanced Classes

Last update: March 5, 2020

Background

Direct marketing, either through mail, email, phone, etc., is a common tactic to acquire customers. Because resources and a customer's attention is limited, the goal is to only target the subset of prospects who are likely to engage with a specific offer. Predicting those potential customers based on readily available information like demographics, past interactions, and environmental factors is a common machine learning problem.

This notebook will train a model which can be used to predict if a customer will enroll for a term deposit at a bank, after one or more phone calls. Hyperparameter tuning will be used in order to try multiple hyperparameter settings and produce the best model.

We will use SageMaker Python SDK, a high level SDK, to simplify the way we interact with SageMaker Hyperparameter Tuning.

Preparation

Let's start by specifying:

- The S3 bucket and prefix that you want to use for training and model data. We'll use the default bucket and create it if it doesn't exist.
- The IAM role used to give training access to your data.

In [1]:
```
import sagemaker
import boto3
import os

bucket = sagemaker.Session().default_bucket()
prefix = 'sagemaker/DEMO-hpo-xgboost-dm'

# Role when working on a notebook instance
role = sagemaker.get_execution_role()
```

Lab 1: Preparing the data

Downloading the data set

Let's start by downloading the [direct marketing dataset (https://archive.ics.uci.edu/ml/datasets/bank+marketing)](https://archive.ics.uci.edu/ml/datasets/bank+marketing) from UCI's ML Repository.

In [2]:
```
!wget -N https://archive.ics.uci.edu/ml/machine-learning-databases/00222/bank-additional.zip
!unzip -o bank-additional.zip
```

```
--2020-03-05 19:27:11--  https://archive.ics.uci.edu/ml/machine-learning-databases/00222/bank-additional.zip
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:443... connected.
HTTP request sent, awaiting response... 304 Not Modified
File ‘bank-additional.zip’ not modified on server. Omitting download.

Archive:  bank-additional.zip
  inflating: bank-additional/.DS_Store
  inflating: __MACOSX/bank-additional/._.DS_Store
  inflating: bank-additional/.Rhistory
  inflating: bank-additional/bank-additional-full.csv
  inflating: bank-additional/bank-additional-names.txt
  inflating: bank-additional/bank-additional.csv
  inflating: __MACOSX/._bank-additional
```

In [3]:
```
!head ./bank-additional/bank-additional-full.csv
```

```
"age";"job";"marital";"education";"default";"housing";"loan";"contact";"month";"day_of_week";"duration";"campaign";"pdays";"previous";"poutcome";"emp.var.rate";"cons.price.idx";"cons.conf.idx";"euribor3m";"nr.employed";"y"
56;"housemaid";"married";"basic.4y";"no";"no";"no";"telephone";"may";"mon";261;1;999;0;"nonexistent";1.1;93.994;-36.4;4.857;5191;"no"
57;"services";"married";"high.school";"unknown";"no";"no";"telephone";"may";"mon";149;1;999;0;"nonexistent";1.1;93.994;-36.4;4.857;5191;"no"
37;"services";"married";"high.school";"no";"yes";"no";"telephone";"may";"mon";226;1;999;0;"nonexistent";1.1;93.994;-36.4;4.857;5191;"no"
40;"admin.";"married";"basic.6y";"no";"no";"no";"telephone";"may";"mon";151;1;999;0;"nonexistent";1.1;93.994;-36.4;4.857;5191;"no"
56;"services";"married";"high.school";"no";"no";"yes";"telephone";"may";"mon";307;1;999;0;"nonexistent";1.1;93.994;-36.4;4.857;5191;"no"
45;"services";"married";"basic.9y";"unknown";"no";"no";"telephone";"may";"mon";198;1;999;0;"nonexistent";1.1;93.994;-36.4;4.857;5191;"no"
59;"admin.";"married";"professional.course";"no";"no";"no";"telephone";"may";"mon";139;1;999;0;"nonexistent";1.1;93.994;-36.4;4.857;5191;"no"
41;"blue-collar";"married";"unknown";"unknown";"no";"no";"telephone";"may";"mon";217;1;999;0;"nonexistent";1.1;93.994;-36.4;4.857;5191;"no"
24;"technician";"single";"professional.course";"no";"yes";"no";"telephone";"may";"mon";380;1;999;0;"nonexistent";1.1;93.994;-36.4;4.857;5191;"no"
```

We need to load this CSV file, inspect it, pre-process it, etc. Please don't write custom Python code to do this!

Instead, developers typically use libraries such as:

- Pandas: a library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language: https://pandas.pydata.org/ (https://pandas.pydata.org/).
- Numpy: a fundamental package for scientific computing with Python: http://www.numpy.org/ (http://www.numpy.org/)

Along the way, we'll use functions from these two libraries. You should definitely become familiar with them, they will make your life much easier when working with large datasets.

In [4]:
```
import numpy as np  # For matrix operations and numerical processing
import pandas as pd # For munging tabular data
```

Let's read the CSV file into a Pandas data frame and take a look at the first few lines.

In [5]:
```
# https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html
data = pd.read_csv('./bank-additional/bank-additional-full.csv', sep=';')
pd.set_option('display.max_columns', 500)     # Make sure we can see all of th
e columns
pd.set_option('display.max_rows', 50)         # Keep the output on one page
data[:10] # Show the first 10 lines
```

Out[5]:

	age	job	marital	education	default	housing	loan	contact	month	day_o
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	
1	57	services	married	high.school	unknown	no	no	telephone	may	
2	37	services	married	high.school	no	yes	no	telephone	may	
3	40	admin.	married	basic.6y	no	no	no	telephone	may	
4	56	services	married	high.school	no	no	yes	telephone	may	
5	45	services	married	basic.9y	unknown	no	no	telephone	may	
6	59	admin.	married	professional.course	no	no	no	telephone	may	
7	41	blue-collar	married	unknown	unknown	no	no	telephone	may	
8	24	technician	single	professional.course	no	yes	no	telephone	may	
9	25	services	single	high.school	no	yes	no	telephone	may	

You can see below the (amount of lines, number of column) for the entire dataset.

In [6]:
```
data.shape # (number of lines, number of columns)
```

Out[6]:
```
(41188, 21)
```

Specifics on each of the features:

Demographics:

- `age` : Customer's age (numeric)
- `job` : Type of job (categorical: 'admin.', 'services', ...)
- `marital` : Marital status (categorical: 'married', 'single', ...)
- `education` : Level of education (categorical: 'basic.4y', 'high.school', ...)

Past customer events:

- `default` : Has credit in default? (categorical: 'no', 'unknown', ...)
- `housing` : Has housing loan? (categorical: 'no', 'yes', ...)
- `loan` : Has personal loan? (categorical: 'no', 'yes', ...)

Past direct marketing contacts:

- `contact` : Contact communication type (categorical: 'cellular', 'telephone', ...)
- `month` : Last contact month of year (categorical: 'may', 'nov', ...)
- `day_of_week` : Last contact day of the week (categorical: 'mon', 'fri', ...)
- `duration` : Last contact duration, in seconds (numeric). Important note: If duration = 0 then `y` = 'no'.

Campaign information:

- `campaign` : Number of contacts performed during this campaign and for this client (numeric, includes last contact)
- `pdays` : Number of days that passed by after the client was last contacted from a previous campaign (numeric)
- `previous` : Number of contacts performed before this campaign and for this client (numeric)
- `poutcome` : Outcome of the previous marketing campaign (categorical: 'nonexistent','success', ...)

External environment factors:

- `emp.var.rate` : Employment variation rate - quarterly indicator (numeric)
- `cons.price.idx` : Consumer price index - monthly indicator (numeric)
- `cons.conf.idx` : Consumer confidence index - monthly indicator (numeric)
- `euribor3m` : Euribor is short for Euro Interbank Offered Rate. The Euribor rates are based on the average interest rates at which a large panel of European banks borrow funds from one another. This is the Euribor 3 month rate - daily indicator (numeric)
- `nr.employed` : Number of employees - quarterly indicator (numeric)

Target variable:

- `y` : Has the client subscribed a term deposit? (binary: 'yes','no')

Let's look at the 2 classes of data we have

In [7]:
```
one_class = data[data['y']=='yes']
one_class_count = one_class.shape[0]
print("Positive samples: %d" % one_class_count)

zero_class = data[data['y']=='no']
zero_class_count = zero_class.shape[0]
print("Negative samples: %d" % zero_class_count)

zero_to_one_ratio = zero_class_count/one_class_count
print("Ratio: %.2f" % zero_to_one_ratio)
```

```
Positive samples: 4640
Negative samples: 36548
Ratio: 7.88
```

The two classes are extremely unbalanced as you can see with the ratio of positive versus negative and it could be a problem for our classifier.

Let's talk about the data. At a high level, we can see:

- We have a little over 40K customer records, 20 features plus a target variable ('y') for each customer
- The features are mixed; some numeric, some categorical
- The data appears to be sorted, at least by `time` and `contact`, maybe more

Transforming the dataset

Cleaning up data is part of nearly every machine learning project. It arguably presents the biggest risk if done incorrectly and is one of the more subjective aspects in the process. Several common techniques include:

- Handling missing values: Some machine learning algorithms are capable of handling missing values, but most would rather not. Options include:
  - Removing observations with missing values: This works well if only a very small fraction of observations have incomplete information.
  - Removing features with missing values: This works well if there are a small number of features which have a large number of missing values.
  - Imputing missing values: Entire books (https://www.amazon.com/Flexible-Imputation-Missing-Interdisciplinary-Statistics/dp/1439868247) have been written on this topic, but common choices are replacing the missing value with the mode or mean of that column's non-missing values.
- Converting categorical to numeric: The most common method is one hot encoding, which for each feature maps every distinct value of that column to its own feature which takes a value of 1 when the categorical feature is equal to that value, and 0 otherwise.
- Oddly distributed data: Although for non-linear models like Gradient Boosted Trees, this has very limited implications, parametric models like linear regression can produce wildly inaccurate estimates when fed highly skewed data. In some cases, simply taking the natural log of the features is sufficient to produce more normally distributed data. In other words, bucketing values into discrete ranges is helpful. These buckets can then be treated as categorical variables and included in the model when one hot encoded.
- Handling more complicated data types: Mainpulating images, text, or data at varying grains.

Luckily, some of these aspects have already been handled for us, and the algorithm we are showcasing tends to do well at handling sparse or oddly distributed data. Therefore, let's keep pre-processing simple.

First of all, many records have the value of "999" for pdays, number of days that passed by after a client was last contacted. It is very likely to be a magic number to represent that no contact was made before. Considering that, we create a new column called "no_previous_contact", then grant it value of "1" when pdays is 999 and "0" otherwise.

In [8]:
```
[np.min(data['pdays']), np.max(data['pdays'])]
```

Out[8]:
```
[0, 999]
```

In [9]:
```
# Indicator variable to capture when pdays takes a value of 999
# https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.where.html
data['no_previous_contact'] = np.where(data['pdays'] == 999, 1, 0)
data = data.drop(['pdays'], axis=1)
```

In the "job" column, there are categories that mean the customer is not working, e.g., "student", "retire", and "unemployed". Since it is very likely whether or not a customer is working will affect his/her decision to enroll in the term deposit, we generate a new column to show whether the customer is working based on "job" column.

In [10]:
```
data['job'].value_counts()
```

Out[10]:
```
admin.           10422
blue-collar       9254
technician        6743
services          3969
management        2924
retired           1720
entrepreneur      1456
self-employed     1421
housemaid         1060
unemployed        1014
student            875
unknown            330
Name: job, dtype: int64
```

In [11]:
```
# Indicator for individuals not actively employed
# https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.in1d.html
data['not_working'] = np.where(np.in1d(data['job'], ['student', 'retired', 'unemployed']), 1, 0)
```

Last but not the least, we convert categorical to numeric, as is suggested above.

In [12]:
```
# https://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html
model_data = pd.get_dummies(data)  # Convert categorical variables to sets of indicators
model_data[:10]
```

Out[12]:

	age	duration	campaign	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.e
0	56	261	1	0	1.1	93.994	-36.4	4.857	
1	57	149	1	0	1.1	93.994	-36.4	4.857	
2	37	226	1	0	1.1	93.994	-36.4	4.857	
3	40	151	1	0	1.1	93.994	-36.4	4.857	
4	56	307	1	0	1.1	93.994	-36.4	4.857	
5	45	198	1	0	1.1	93.994	-36.4	4.857	
6	59	139	1	0	1.1	93.994	-36.4	4.857	
7	41	217	1	0	1.1	93.994	-36.4	4.857	
8	24	380	1	0	1.1	93.994	-36.4	4.857	
9	25	50	1	0	1.1	93.994	-36.4	4.857	

As you can see, each categorical column (job, marital, education, etc.) has been replaced by a set of new columns, one for each possible value in the category. Accordingly, we now have 66 columns instead of 21.

In [13]:
```
model_data.shape
```

Out[13]:
```
(41188, 66)
```

Selecting features

Another question to ask yourself before building a model is whether certain features will add value in your final use case. For example, if your goal is to deliver the best prediction, then will you have access to that data at the moment of prediction? Knowing it's raining is highly predictive for umbrella sales, but forecasting weather far enough out to plan inventory on umbrellas is probably just as difficult as forecasting umbrella sales without knowledge of the weather. So, including this in your model may give you a false sense of precision.

Following this logic, let's remove the economic features and `duration` from our data as they would need to be forecasted with high precision to use as inputs in future predictions.

Even if we were to use values of the economic indicators from the previous quarter, this value is likely not as relevant for prospects contacted early in the next quarter as those contacted later on.

In [14]:
```
# https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html
model_data = model_data.drop(['duration', 'emp.var.rate', 'cons.price.idx', 'cons.conf.idx', 'euribor3m', 'nr.employed'], axis=1)
```

In [15]:
```
model_data.shape
```

Out[15]:
```
(41188, 60)
```

Splitting the dataset

We'll then split the dataset into training (70%), validation (20%), and test (10%) datasets and convert the datasets to the right format the algorithm expects. We will use training and validation datasets during training and we will try to maximize the accuracy on the validation dataset.

Once the model has been deployed, we'll use the test dataset to evaluate its performance.

Amazon SageMaker's XGBoost algorithm expects data in the libSVM or CSV data format. For this example, we'll stick to CSV. Note that the first column must be the target variable and the CSV should not include headers. Also, notice that although repetitive it's easiest to do this after the train|validation|test split rather than before. This avoids any misalignment issues due to random reordering.

In [16]:
```
# Set the seed to 123 for reproductibility
# https://pandas.pydata.org/pandas-docs/version/0.22/generated/pandas.DataFrame.sample.html
# https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.split.html
train_data, validation_data, test_data = np.split(model_data.sample(frac=1, random_state=123),
                                                  [int(0.7 * len(model_data)), int(0.9*len(model_data))])

# Drop the two columns for 'yes' and 'no' and add 'yes' back as first column of the dataframe
# https://pandas.pydata.org/pandas-docs/stable/generated/pandas.concat.html
pd.concat([train_data['y_yes'], train_data.drop(['y_no', 'y_yes'], axis=1)], axis=1).to_csv('train.csv', index=False, header=False)
pd.concat([validation_data['y_yes'], validation_data.drop(['y_no', 'y_yes'], axis=1)], axis=1).to_csv('validation.csv', index=False, header=False)
```

In [17]:
```
!ls -l *.csv
```

```
-rw-rw-r-- 1 ec2-user ec2-user 3431668 Mar  5 19:27 train.csv
-rw-rw-r-- 1 ec2-user ec2-user  980538 Mar  5 19:27 validation.csv
```

Now we'll copy the files to S3 for Amazon SageMaker training to pickup.

In [18]:
```
boto3.Session().resource('s3').Bucket(bucket).Object(os.path.join(prefix, 'train/train.csv')).upload_file('train.csv')
boto3.Session().resource('s3').Bucket(bucket).Object(os.path.join(prefix, 'validation/validation.csv')).upload_file('validation.csv')
```

SageMaker needs to know where the training and validation sets are located, so let's define that and create a dictionary `s3_data` of those.

In [19]:
```
s3_input_train = sagemaker.s3_input(s3_data='s3://{}/{}/train'.format(bucket, prefix), content_type='csv')
s3_input_validation = sagemaker.s3_input(s3_data='s3://{}/{}/validation/'.format(bucket, prefix), content_type='csv')

s3_data = {'train': s3_input_train, 'validation': s3_input_validation}
```

Lab 2: Training our first model with XGBoost

The problem we're trying to solve is a classification problem: will a given customer react positively to our marketing offer or not? In order to answer this question, let's train a classification model with XGBoost, a popular open source algorithm available in SageMaker.

You can read more about the XGBoost algorithm in its documentation:
https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html
(https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html)

SageMaker uses containers to execute trainings on. So the first step is to get the URL of the built-in XGBoost container image that is hosted in the Amazon Elastic Container Registry, a fully-managed Docker container registry. Since there are 2 versions of that container, we specify the newer version.

The easiest way to execute a training within SageMaker is to use an Estimator object. It's a high level interface for SageMaker training. Once it's initialized with the appropriate parameters, you can start a SageMaker training. You can look at the documentation here (https://sagemaker.readthedocs.io/en/latest/estimators.html).

- First, we specify the container image for XGBoost.
- Then we must specify a role as SageMaker will need to download the data from S3 and upload the data back to S3.
- The amount and type of instances is then specified. Amazon SageMaker XGBoost 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).
- The input type is set to File as it's the only one supported by XGBoost. This is as opposed to Pipe mode which wouldn't require to download the entirety of the data on the instance to start the training. Since our data isn't made of multiple gigabytes, there wouldn't be any differences even if it supported it.
- We then specify where to place the output of the model in S3 by specifying the bucket and the prefix we defined at the beginning of this notebook.
- Finally, the session variable is specified that contains the credentials we are currently using in this notebook to execute the API calls for starting the training within SageMaker.

In [20]:
```
from sagemaker.amazon.amazon_estimator import get_image_uri

sess = sagemaker.Session()

region = boto3.Session().region_name
container = get_image_uri(region, 'xgboost', '0.90-1')

xgb = sagemaker.estimator.Estimator(container,
                                    role,
                                    train_instance_count=1,
                                    train_instance_type='ml.m5.xlarge',
                                    input_mode="File",
                                    output_path='s3://{}/{}/output'.format(buc
ket, prefix),
                                    sagemaker_session=sess)
```

Setting hyper parameters

Each built-in algorithm has a set of hyperparameters. Here are the ones for XGBoost:
https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost_hyperparameters.html
(https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost_hyperparameters.html)

That probably looks a little weird :) Let's stick to the two **required** parameters:

- Build a binary classifier: 'binary:logistic'. We want our prediction to be a classification as either yes (1) or no (0), so that's a binary classifier.
- Train for 100 rounds (but is this the right value? More on this later).

In [21]:
```
xgb.set_hyperparameters(objective='binary:logistic', num_round=100)
```

We're all set. Let's train! We use the SageMaker Estimator fit() function
(https://sagemaker.readthedocs.io/en/stable/estimators.html#sagemaker.estimator.EstimatorBase.fit) passing in the dictionary we created for the training and validation data. While the job is running, head out to the AWS Management Console where you opened this notebook and look under the **Training jobs** section. Select your training job, look at hyperparameters, etc. In the "Monitor" subsection, you can also view the **training log**, as well as the **training metrics** in CloudWatch. The dashboard may not show anything for a few minutes while they are computed.

The training should take less than 5 minutes.

In [22]:
```
xgb.fit(s3_data)
```

```
2020-03-05 19:27:16 Starting - Starting the training job...
2020-03-05 19:27:20 Starting - Launching requested ML instances...
2020-03-05 19:28:15 Starting - Preparing the instances for training.........
2020-03-05 19:29:41 Downloading - Downloading input data
2020-03-05 19:29:41 Training - Training image download completed. Training in
progress..INFO:sagemaker-containers:Imported framework sagemaker_xgboost_cont
ainer.training
INFO:sagemaker-containers:Failed to parse hyperparameter objective value bina
ry:logistic to Json.
Returning the value itself
INFO:sagemaker-containers:No GPUs detected (normal if no gpus installed)
INFO:sagemaker_xgboost_container.training:Running XGBoost Sagemaker in algori
thm mode
INFO:root:Determined delimiter of CSV input is ','
INFO:root:Determined delimiter of CSV input is ','
INFO:root:Determined delimiter of CSV input is ','
[19:29:43] 28831x58 matrix with 1672198 entries loaded from /opt/ml/input/dat
a/train?format=csv&label_column=0&delimiter=,
INFO:root:Determined delimiter of CSV input is ','
[19:29:43] 8238x58 matrix with 477804 entries loaded from /opt/ml/input/data/
validation?format=csv&label_column=0&delimiter=,
INFO:root:Single node training.
INFO:root:Train matrix has 28831 rows
INFO:root:Validation matrix has 8238 rows
[0]#011train-error:0.098262#011validation-error:0.101967
[1]#011train-error:0.097603#011validation-error:0.10136
[2]#011train-error:0.096944#011validation-error:0.101602
[3]#011train-error:0.097222#011validation-error:0.100267
[4]#011train-error:0.09684#011validation-error:0.100146
[5]#011train-error:0.097152#011validation-error:0.100146
[6]#011train-error:0.097152#011validation-error:0.100631
[7]#011train-error:0.09632#011validation-error:0.10051
[8]#011train-error:0.096181#011validation-error:0.100753
[9]#011train-error:0.095973#011validation-error:0.100874
[10]#011train-error:0.0958#011validation-error:0.100995
[11]#011train-error:0.09573#011validation-error:0.100874
[12]#011train-error:0.095904#011validation-error:0.100753
[13]#011train-error:0.094516#011validation-error:0.100874
[14]#011train-error:0.094863#011validation-error:0.101967
[15]#011train-error:0.094239#011validation-error:0.101481
[16]#011train-error:0.093892#011validation-error:0.101602
[17]#011train-error:0.093753#011validation-error:0.101724
[18]#011train-error:0.093545#011validation-error:0.101602
[19]#011train-error:0.093268#011validation-error:0.101481
[20]#011train-error:0.093198#011validation-error:0.10136
[21]#011train-error:0.09299#011validation-error:0.101481
[22]#011train-error:0.09306#011validation-error:0.101117
[23]#011train-error:0.09299#011validation-error:0.100995
[24]#011train-error:0.092713#011validation-error:0.10136
[25]#011train-error:0.092331#011validation-error:0.101481
[26]#011train-error:0.092262#011validation-error:0.101602
[27]#011train-error:0.092088#011validation-error:0.101481
[28]#011train-error:0.091846#011validation-error:0.100995
[29]#011train-error:0.091533#011validation-error:0.101238
[30]#011train-error:0.09136#011validation-error:0.101845
[31]#011train-error:0.090736#011validation-error:0.102088
[32]#011train-error:0.090874#011validation-error:0.101967
```

[33]#011train-error:0.090805#011validation-error:0.102209
[34]#011train-error:0.090666#011validation-error:0.101967
[35]#011train-error:0.090458#011validation-error:0.102573
[36]#011train-error:0.090285#011validation-error:0.102331
[37]#011train-error:0.09025#011validation-error:0.102573
[38]#011train-error:0.090042#011validation-error:0.102938
[39]#011train-error:0.089487#011validation-error:0.103423
[40]#011train-error:0.08966#011validation-error:0.103545
[41]#011train-error:0.089175#011validation-error:0.103302
[42]#011train-error:0.089105#011validation-error:0.103423
[43]#011train-error:0.089071#011validation-error:0.103545
[44]#011train-error:0.088551#011validation-error:0.103423
[45]#011train-error:0.088134#011validation-error:0.10318
[46]#011train-error:0.08803#011validation-error:0.103302
[47]#011train-error:0.087614#011validation-error:0.103666
[48]#011train-error:0.087337#011validation-error:0.103059
[49]#011train-error:0.086886#011validation-error:0.103059
[50]#011train-error:0.08699#011validation-error:0.103059
[51]#011train-error:0.086712#011validation-error:0.102816
[52]#011train-error:0.086608#011validation-error:0.102573
[53]#011train-error:0.086539#011validation-error:0.102695
[54]#011train-error:0.086469#011validation-error:0.10318
[55]#011train-error:0.086123#011validation-error:0.103787
[56]#011train-error:0.085984#011validation-error:0.104151
[57]#011train-error:0.085637#011validation-error:0.104273
[58]#011train-error:0.085776#011validation-error:0.104758
[59]#011train-error:0.085602#011validation-error:0.104516
[60]#011train-error:0.085464#011validation-error:0.104637
[61]#011train-error:0.085429#011validation-error:0.104516
[62]#011train-error:0.084909#011validation-error:0.104394
[63]#011train-error:0.084423#011validation-error:0.104516
[64]#011train-error:0.084423#011validation-error:0.104516
[65]#011train-error:0.084388#011validation-error:0.104516
[66]#011train-error:0.084388#011validation-error:0.10488
[67]#011train-error:0.084284#011validation-error:0.10488
[68]#011train-error:0.08425#011validation-error:0.10488
[69]#011train-error:0.083937#011validation-error:0.104758
[70]#011train-error:0.083903#011validation-error:0.104637
[71]#011train-error:0.083695#011validation-error:0.104758
[72]#011train-error:0.08366#011validation-error:0.104273
[73]#011train-error:0.083417#011validation-error:0.104516
[74]#011train-error:0.083348#011validation-error:0.104516
[75]#011train-error:0.083417#011validation-error:0.104151
[76]#011train-error:0.083487#011validation-error:0.104394
[77]#011train-error:0.083105#011validation-error:0.10403
[78]#011train-error:0.08307#011validation-error:0.104273
[79]#011train-error:0.08307#011validation-error:0.104273
[80]#011train-error:0.082862#011validation-error:0.104273
[81]#011train-error:0.082862#011validation-error:0.104273
[82]#011train-error:0.082793#011validation-error:0.104516
[83]#011train-error:0.082585#011validation-error:0.104273
[84]#011train-error:0.082446#011validation-error:0.104516
[85]#011train-error:0.082377#011validation-error:0.104151
[86]#011train-error:0.081856#011validation-error:0.103787
[87]#011train-error:0.081822#011validation-error:0.10403
[88]#011train-error:0.081752#011validation-error:0.103666
[89]#011train-error:0.08144#011validation-error:0.103787

```
[90]#011train-error:0.081093#011validation-error:0.103302
[91]#011train-error:0.080885#011validation-error:0.10318
[92]#011train-error:0.080955#011validation-error:0.103302
[93]#011train-error:0.080781#011validation-error:0.10318
[94]#011train-error:0.080712#011validation-error:0.10318
[95]#011train-error:0.08033#011validation-error:0.103423
[96]#011train-error:0.080053#011validation-error:0.102816
[97]#011train-error:0.079324#011validation-error:0.103059
[98]#011train-error:0.079255#011validation-error:0.103423
[99]#011train-error:0.079116#011validation-error:0.103545

2020-03-05 19:29:54 Uploading - Uploading generated training model
2020-03-05 19:29:54 Completed - Training job completed
Training seconds: 32
Billable seconds: 32
```

Follow-up from training

- What's the training accuracy? The validation accuracy? Write them down, we'll need them later on.
- Would training on more than one instance improve our model?

Lab 3: Deploying our model

Now let's deploy our model to an HTTPS endpoint. All it takes is one line of code. We only need to call the `deploy()` function of our Estimator specifying the amount of instances and the type. Again, we don't need a GPU for this and a General Purpose type instance is totally fine. By now you should have realized that the Estimator object seems to be doing everything here. You use `fit()` to start the training and `deploy()` to deploy the model you trained. The SageMaker SDK tries to make it as easy as possible. The `deploy()` command will create a **Model** from the training job, an **Endpoint configuration** and an **Endpoint**.

While deployment takes place, head out to the AWS Management Console of SageMaker and familiarize yourself with the **Models**, **Endpoint configurations** and **Endpoints** section.

The deployment should take less than 10 minutes. Make sure you to take a look if any errors are displayed from time to time. It would be a good time to dive deeper in the documentation linked in the previous steps, ask questions to your facilitators or take a well deserved break.

In [23]:
```
xgb_endpoint = xgb.deploy(initial_instance_count = 1, instance_type = 'ml.m5.xlarge')
```

```
-----------!
```

Lab 4: Predicting with our model

Earlier, we split our data into training (70%), validation (20%) and test (10%). It's time to use the *test_data* to invoke our model. If you did some quick maths earlier when we split the data, you would know that there are 4,119 records in our *test_data*. That's a bit too much data to send as one request. Note that we could use a SageMaker Batch Transforms job to do this for us which would take care of deploying an endpoint, send all the data to it, store it in S3 and shutdown the endpoint. However, that's not at all what we want to do here. We want to invoke the endpoint directly, so keep reading!

The test_data we created earlier at the time of splitting the original data is currently stored as a NumPy array in the memory of our notebook instance. The SageMaker endpoint we just created is hosting an HTTP API endpoint that is expecting an HTTP POST request. To send test_data as an HTTP POST request, we'll serialize/transform it as a CSV string. That's the default that the built-in XGBoost container is expecting. This data can then be passed to the `predict()` function of our SageMaker endpoint.

Luckily for us, that's just a matter of setting the serializer of our endpoint as *csv_serializer*. Isn't this SDK thing great?

In [24]:
```
from sagemaker.predictor import csv_serializer

xgb_endpoint.content_type = 'text/csv'
xgb_endpoint.serializer = csv_serializer
```

First, we also need to remove the *y_no* and *y_yes* column as that's the label and we definitely don't want to send the result of our prediction to our predictor, it's its just to do that!

In [25]:
```
test_data_no_prediction = test_data.drop(['y_no', 'y_yes'], axis=1).values
test_data_no_prediction.shape
```

Out[25]:
```
(4119, 58)
```

If you remember, we have 10% of the 41,188 rows from our original dataset in the `test_data` function.

In [26]:
```
test_data_no_prediction.shape
```

Out[26]:
```
(4119, 58)
```

Now that we have the data, we can see above that there is a 4,119 lines. That's a lot of data to send to the endpoint directly in one resquest. So instead, we will split the data to send the data 500 rows at a time using the numpy.array_split (https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.array_split.html) function.

In [27]:
```
rows = 500
split_test_data_array = np.array_split(test_data_no_prediction, int(test_data_
no_prediction.shape[0] / float(rows) + 1))
```

Now that we have our data in mini-batches of 500 rows, it's time to loop over each of those and do the following for each of them:

- Use the *xgb_endpoint* we created using the `deploy()` function of our Estimator to execute a prediction using the `predict()` function
- Decode the response as utf-8. The data from the response will be in CSV format
- Join the result to the previous mini-batches results

Finally, we need to convert the predictions that are in a string format into a NumPy array to make it easier to work with than a string.

In [28]:
```
predictionsResult = ''
for test_data_500rows in split_test_data_array:
    predictionsResult = ','.join([predictionsResult, xgb_endpoint.predict(test
_data_500rows).decode('utf-8')])

predictions = np.fromstring(predictionsResult[1:], sep=',')
```

Looking at that data, we can see that we have an array the same size as our `test_data` .

In [29]:
```
print(predictions)
predictions.shape
```

```
[0.0702678  0.07871322 0.05670157 ... 0.11874407 0.07205555 0.04528835]
```

Out[29]:
```
(4119,)
```

For each sample, our binary classifier returns a probability between 0 and 1. Since we decided to maximize accuracy, the model sets a threshold of 0.5: anything lower is treated as a 0, anything higher as a 1.

To dive a little deeper: the threshold is baked in the metric that XGBoost uses. Here, we use the default 'eval_metric' for classification, i.e. 'error'. This metric has a default threshold of 0.5. If you look at the XGBoost doc (https://xgboost.readthedocs.io/en/latest/parameter.html), you'll see that it's possible to pass a different threshold, doing something like: `xgb.set_hyperparameters(objective='binary:logistic', num_round=100, eval_metric='error@0.2')`

So we first need to round up (1) or down (0) the predictions based on the threshold of 0.5.

Then we use the crosstab function (https://pandas.pydata.org/pandas-docs/version/0.22/generated/pandas.crosstab.html) from pandas to get a confusion matrix by computing a frequency table of the index, the actual value, versus the columns, the predicted value. We also pass two arguments to label the index (rows) and columns.

In [30]:
```
rounded_predictions = np.where(predictions > 0.5, 1, 0)

# Also called a 'confusion matrix'
pd.crosstab(index=test_data['y_yes'],
            columns=rounded_predictions,
            rownames=['truth'], colnames=['predictions'])
```

Out[30]:

truth \ predictions	0	1
0	3567	71
1	355	126

How well did we do on the test set (**your own numbers probably vary**):

- 3567 true negatives (top left) were correctly predicted. This means that we correctly predicted that calling the customer was not going to help get them enrolled and it actually didn't.
- 126 true positives (bottom right) were correctly predicted. This means that we correctly predicted that calling the customer was going to get them enrolled and it actually did.
- 355 positives were incorrectly predicted as negatives (false negatives, bottom left), so we'll probably miss business opportunities by not engaging with these customers. It looks like this model is too conservative!
- 71 negatives were incorrectly predicted as positives (false positives, top right), so we'll probably waste our time engaging with these customers.

All in all, our accuracy is: (3567+126)/(3567+355+71+126)=0.8966 aka 89.66%. This is consistent with the validation accuracy we observed during the training process. If it wasn't, then it would mean that our validation set and test set have different distributions. That would be a major problem and we would definitely have to fix the way they're built.

Other useful metrics for classifiers are precision, recall and F1 score (https://en.wikipedia.org/wiki/F1_score):

- **Precision**: the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is the ability of the classifier not to label as positive a sample that is negative.
- **Recall**: the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is the ability of the classifier to find all the positive samples.
- **F1 score**: a weighted mean of precision and recall. 1 is the best possible score and 0 is the worst.

We are going to use the `precision_recall_fscore_support()` function (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html) from the sickit learn library to get this data without having to manually do it ourselves. We pass in the ground truth (correct) target values and the estimated targets that were returned by the predictor. We also set the average as binary as we have a binary classifier.

In [31]:
```
from sklearn.metrics import precision_recall_fscore_support
score = precision_recall_fscore_support(
            test_data['y_yes'], rounded_predictions, average='binary')

# Precision, recall, F1 score
print(score)
```

```
(0.6395939086294417, 0.26195426195426197, 0.37168141592920356, None)
```

These false negatives are really hurting recall, which in turns impacts the F1 score. As we saw above, we decided to optimize for accuracy, and this comes at the expense of a rather poor F1 score. If our goal was to maximize the F1 score, we could decide on a different threshold. Keep in mind that you can either optimize false positives or false negatives, but not both. You have to decide which ones have the bigger impact on your application.

This trade-off is made more difficult by the class imbalance problem. Fixing it is beyond the scope of this workshop, but we could use techniques like:

- adding real data to the positive class (real or synthetic data),
- adding synthetic data to the positive class (e.g. over-sampling),
- using the 'scale_pos_weight' hyper parameter to account for class imbalance,
- more pre-processing, more feature engineering, etc.

If you want to dive deeper, [this blog post (https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/)](https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/) is a good starting point.

Deleting the endpoint

Once that we're done predicting, we can delete the endpoint (and stop paying for it), the model and the endpoint configuration.

In [32]:
```
xgb_endpoint.delete_model()
xgb_endpoint.delete_endpoint()
```

All this is pretty cool (hopefully) but how can we get a better model?

In our first attempt, we used the minimum number of hyperparameters. Surely we can tweak a little more and improve the accuracy :)

How long should we train for?

In the previous example, we set the number of rounds to 100. How do we know if this is the right value or not? If we don't train long enough, we could be missing out on accuracy. If we train for too long, we could be overfitting... or just wasting time and money.

XGBoost has an hyper parameter named *early_stopping_rounds*. It stops training when accuracy hasn't improved in *early_stopping_rounds* rounds. Take a minute to read the doc (https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost_hyperparameters.html).

Think about what you would need to do to try this...

If you answered, changing the Estimator and then doing a `fit()` again and comparing the validation error, that's one way to do it. You could also deploy that model to the endpoint again and see if it gives better results.

Based on the time we have, I'm not asking you to do this.

How deep should the trees be?

Tree depth is obviously an important parameter for tree-based algorithms. XGBoost has an hyper parameter named *max_depth*: by default, it is set to 6. Is this too high? Too low? Maybe we could improve our accuracy by building a more complex model based on a deeper tree. Or maybe the model would generalize better with a shallower tree?

The same steps you just thought about in the previous example would have to be done again here. Again, I'm not asking you to do this based on the time we have.

So, selecting the right value for hyperparameters means you have to do a lot of trial and errors. And what about other hyperparameters? **We can't reasonably keep guessing like this**.

Fortunately, SageMaker supports Automatic Model Tuning.

Lab 5: Finding optimal hyperparameters with Automatic Model Tuning

Understanding Automatic Model Tuning

We will use SageMaker automatic model tuning to automate the searching process effectively. Specifically, we specify a range, or a list of possible values in the case of categorical hyperparameters, for each of the hyperparameter that we plan to tune. SageMaker hyperparameter tuning will automatically launch multiple training jobs with different hyperparameter settings, evaluate results of those training jobs based on a predefined "objective metric", and select the hyperparameter settings for future attempts based on previous results. For each hyperparameter tuning job, we will give it a budget (maximum number of training jobs) and it will complete once that many training jobs have been executed.

We will tune four hyperparameters in this example. Don't worry if this sounds over-complicated, we don't need to understand this in detail right now.

- *eta*: 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.
- *alpha*: L1 regularization term on weights. Increasing this value makes models more conservative.
- *min_child_weight*: Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, the building process gives up further partitioning. In linear regression models, this simply corresponds to a minimum number of instances needed in each node. The larger the algorithm, the more conservative it is.
- *max_depth*: Maximum depth of a tree. Increasing this value makes the model more complex and likely to be overfitted.

In [33]:
```
from sagemaker.tuner import IntegerParameter, ContinuousParameter

hyperparameter_ranges = {'eta': ContinuousParameter(0, 1),
                         'min_child_weight': ContinuousParameter(1, 10),
                         'alpha': ContinuousParameter(0, 2),
                         'max_depth': IntegerParameter(1, 10)
                         }
```

We first configure the training jobs the hyperparameter tuning job will launch by initiating an estimator the same way we've done it previously.

In [34]:
```
xgb1 = sagemaker.estimator.Estimator(container,
                                     role,
                                     train_instance_count=1,
                                     train_instance_type='ml.m5.xlarge',
                                     input_mode="File",
                                     output_path='s3://{}/{}/output'.format(buc
ket, prefix),
                                     sagemaker_session=sess)
```

Then we will also set the hyperparameters that we don't want to auto tune. We are using that `early_stopping_rounds` attribute so we don't train for too long, but we set our `num_rounds` way higher.

In [35]:
```
xgb1.set_hyperparameters(objective='binary:logistic',
                         num_round=1000,
                         early_stopping_rounds=100)
```

Finally, we will create a *HyperparameterTuner* object, to which we provide:

- The XGBoost estimator
- Objective metric name and type. The objective here is to minimize the validation error. We could have also done that on the f1 score. So every time a new training will occur, SageMaker will pick the next hyperparameters based on the lowest validation error. This objective is something SageMaker can get from the CloudWatch logs. It knows how to get this data by parsing the logs and use a regular expression.
- The hyperparameter ranges we defined above
- The maximum number of training jobs to run. We set it to 9 in this case due to the time in this lab. We would have to find an appropriate number based on the amount of hyperparameters we train on.
- The number of parallel training jobs which we set to 3.

In [36]:
```
tuner = sagemaker.tuner.HyperparameterTuner(estimator=xgb1,
                            objective_metric_name='validation:error',
                            objective_type='Minimize',
                            hyperparameter_ranges=hyperparameter_ranges,
                            max_jobs=10,
                            max_parallel_jobs=2)
```

Launching Automatic Model Tuning

Now we can launch a hyperparameter tuning job by calling the `fit()` function. This function is a little bit different than the previous as instead of making you wait for it to finish, it just returns right away which is something you would call an asynchronous call while earlier, it was a synchronous call.

In [37]:
```
tuner.fit(s3_data)
```

Let's just run a quick check of the tuning job status to make sure it started successfully. You should see an *InProgress* status below.

In [38]:
```
sagemaker_client = boto3.Session().client(service_name='sagemaker')

# Get tuning job name
job_name = tuner.latest_tuning_job.job_name
print(job_name)

sagemaker_client.describe_hyper_parameter_tuning_job(
    HyperParameterTuningJobName=job_name)['HyperParameterTuningJobStatus']
```

```
sagemaker-xgboost-200305-1936
```

Out[38]:
```
'InProgress'
```

Now that the job is in progress, let's make get your notebook to wait for it in case you decided to just run every single cells without looking. We aren't judging :). You can continue reading the next step to know what to do while this finishes.

In [39]:
```
tuner.wait()
```

```
.....................................................................
```

Since the job is launched, you can look at its progress in the AWS Management Console for SageMaker under the *Hyperparameter tuning jobs* section.

This job will run for about 10-15 minutes, so there's time for a break. If you're too hardcore for a break, you can ask questions to your facilitators, continue reading this notebook in case you are running out of time and want to know what will happen next or you can use the time to learn more about the XGBoost algorithm:

- Documentation: https://xgboost.readthedocs.io/en/latest/ (https://xgboost.readthedocs.io/en/latest/)
- Research paper: https://arxiv.org/abs/1603.02754 (https://arxiv.org/abs/1603.02754)

Running the below cell should tell you that 10 training jobs have completed. Although the *waiter* above should take care of it, in case you didn't go in the AWS Management Console to look at it, this will show you that they were completed.

In [40]:
```
tuning_job_result = sagemaker_client.describe_hyper_parameter_tuning_job(Hyper
ParameterTuningJobName=job_name)

status = tuning_job_result['HyperParameterTuningJobStatus']
if status != 'Completed':
    print('Reminder: the tuning job has not been completed.')

job_count = tuning_job_result['TrainingJobStatusCounters']['Completed']
print("%d training jobs have completed" % job_count)
```

```
10 training jobs have completed
```

The following code returns the best training job model name and the hyperparameters used to provide it. Can you see if you were close to the right numbers before?

In [41]:
```
from pprint import pprint
if tuning_job_result.get('BestTrainingJob',None):
    print("Best model found:")
    pprint(tuning_job_result['BestTrainingJob'])
else:
    print("No training jobs have reported results yet.")
```

```
Best model found:
{'CreationTime': datetime.datetime(2020, 3, 5, 19, 47, 43, tzinfo=tzlocal()),
 'FinalHyperParameterTuningJobObjectiveMetric': {'MetricName': 'validation:er
ror',
                                                 'Value': 0.0995389968156814
6},
 'ObjectiveStatus': 'Succeeded',
 'TrainingEndTime': datetime.datetime(2020, 3, 5, 19, 50, 16, tzinfo=tzlocal
()),
 'TrainingJobArn': 'arn:aws:sagemaker:ca-central-1:761424745283:training-job/
sagemaker-xgboost-200305-1936-009-7952a75b',
 'TrainingJobName': 'sagemaker-xgboost-200305-1936-009-7952a75b',
 'TrainingJobStatus': 'Completed',
 'TrainingStartTime': datetime.datetime(2020, 3, 5, 19, 49, 25, tzinfo=tzloca
l()),
 'TunedHyperParameters': {'alpha': '1.9978622881346322',
                          'eta': '0.04423656300480428',
                          'max_depth': '4',
                          'min_child_weight': '2.0823431089720943'}}
```

Now that the tuning job is complete, we can do the deployment.

Optional - Lab 6: Deploying our best 2 models

This is an optional lab based on the time where we ask you to deploy the best 2 models. It's something fictional that you wouldn't really be doing typically. You should be going with the best model. However, this isn't a production deployment :). So we will ask you to deploy the best 2 models to learn a feature of SageMaker that allows you to do canary testing.

The easiest way to deploy the best model is to use the `deploy()` API of the current HyperparameterTuner object similar to how we deployed the model from our training earlier. If we wanted to use a previous tuning job, you would need to use the `attach()` API to attach to it before calling `deploy()` because the current object is pointing to the best. You can find more information here (https://sagemaker.readthedocs.io/en/latest/tuner.html).

However, this isn't what we will do today. Normally it would be, but what if you wanted to deploy the best 2 models to test them? To do that, we will use the A/B testing feature of SageMaker.

SageMaker is letting you deploy multiple variants of a model to the same endpoint. This is useful for testing variations of a model in production. Let's try this and deploy the top 2 models trained by the tuning job.

First, let's figure out what the top 2 jobs are. We will show the completed jobs sorted by descending accuracy.

In [42]:
```
stats = tuner.analytics().dataframe()                # grab job stats in a Pand
as dataframe
stats = stats.nsmallest(2, 'FinalObjectiveValue')    # keep top two performing
 models (lowest validation error)
stats.head()
```

Out[42]:

	FinalObjectiveValue	TrainingElapsedTimeSeconds	TrainingEndTime	TrainingJobName	Training
1	0.099539	51.0	2020-03-05 19:50:16+00:00	sagemaker-xgboost-200305-1936-009-7952a75b	
2	0.100388	48.0	2020-03-05 19:47:47+00:00	sagemaker-xgboost-200305-1936-008-66a6a82b	

Then we take take the name from both of those models and assign them to 2 new variables.

In [43]:
```
top1_model_name = stats.iloc[0]['TrainingJobName']
top2_model_name = stats.iloc[1]['TrainingJobName']
print(top1_model_name)
print(top2_model_name)
```

```
sagemaker-xgboost-200305-1936-009-7952a75b
sagemaker-xgboost-200305-1936-008-66a6a82b
```

We then use the `create_model()` API (https://docs.aws.amazon.com/sagemaker/latest/dg/API_CreateModel.html) to register the two best training jobs as SageMaker models. The reason why you have to do this manually this time is that the `deploy()` API used earlier on the training job did that automatically for you. This time, we are doing it the hard way as we want to have more than one variant.

As we will need to create 2 models and because we are follow best practices as good developers (right?), we create a function that will create the models and call it twice. The `create_model()` API takes the model name that we have from above, the IAM Role from this current notebook instance to be able to execute the call and some information about the primary container to use.

You may be wondering why it's called a primary container. The reason for that is that a SageMaker endpoint can use something called an Inference Pipeline (https://docs.aws.amazon.com/sagemaker/latest/dg/inference-pipelines.html) where you can have up to 5 containers that are invoked sequentially. You use an inference pipeline to define and deploy any combination of pretrained Amazon SageMaker built-in algorithms and your own custom algorithms packaged in Docker containers. You can use an inference pipeline to combine preprocessing, predictions, and post-processing data science tasks. In this case, we only need one container as it's our model, but you can see how this could be useful to you in other occasions.

The definition of the primary container is made up of the same container for XGBoost that we have been using for training and the URL of the model where it's stored in S3. This is found by calling the `describe_training_job()` API.

In [44]:
```
def create_model(model_name):
    model_info = sagemaker_client.describe_training_job(TrainingJobName=model_
name)
    model_data = model_info['ModelArtifacts']['S3ModelArtifacts']
    primary_container = {'Image': container,'ModelDataUrl': model_data}

    create_model_response = sagemaker_client.create_model(
        ModelName = model_name,
        ExecutionRoleArn = role,
        PrimaryContainer = primary_container
    )
    print(create_model_response['ModelArn'])

create_model(top1_model_name)
create_model(top2_model_name)
```

```
arn:aws:sagemaker:ca-central-1:761424745283:model/sagemaker-xgboost-200305-19
36-009-7952a75b
arn:aws:sagemaker:ca-central-1:761424745283:model/sagemaker-xgboost-200305-19
36-008-66a6a82b
```

We then need to define an endpoint configuration with the required infrastructure settings for the endpoint. Again, this was something that the `deploy()` API did for us earlier.

First, we will need to give a name to our endpoint and endpoint config that needs to be unique. That's why we add the timestamp to those names in case you decide to re-run this cell.

Then we create the endpoint configuration using that name and specifying a few parameters. Those parameters are repeated twice as we have 2 models to deploy.

- The same type of instance as we've used previously (ml.m5.xlarge)
- The same amount of instances as before (1)
- The name of the model that we created in the previous command
- The name of the model variant which is arbitrary here, but that helps us understand which one is the best (top1) and second best (top2)
- The weight for each model. Here we could send half the traffic to the top1 and half to the top2, however we wanted to demonstrate to you how you could send 2/3 of the traffic to the best model and only 1/3 to the second best. The reason here is purely for learning purposes.

Finally, we print the name of the endpoint configuration in case you wanted to go look at it in the AWS Management Console under the *Endpoint configurations* section.

In [45]:
```
import time
timestamp = time.strftime('%Y-%m-%d-%H-%M-%S', time.gmtime())
endpoint_name = 'banking-' + timestamp
endpoint_config_name = 'banking-' + timestamp

endpoint_config_response = sagemaker_client.create_endpoint_config(
    EndpointConfigName = endpoint_config_name,
    ProductionVariants=[
        {
            'InstanceType':'ml.m5.xlarge',
            'InitialInstanceCount':1,
            'ModelName':top1_model_name,
            'VariantName':'top1',
            'InitialVariantWeight':2     # two thirds of the traffic
        },
        {
            'InstanceType':'ml.m5.xlarge',
            'InitialInstanceCount':1,
            'ModelName':top2_model_name,
            'VariantName':'top2',
            'InitialVariantWeight':1    # one third of the traffic
        }
    ])

print('Endpoint configuration name: {}'.format(endpoint_config_name))
```

```
Endpoint configuration name: banking-2020-03-05-19-50-41
```

Now, we can deploy the endpoint with the `create_endpoint()` API using the endpoint configuration we created above. The following command will output the ARN of the endpoint.

In [46]:
```
endpoint_params = {
    'EndpointName': endpoint_name,
    'EndpointConfigName': endpoint_config_name,
}
endpoint_response = sagemaker_client.create_endpoint(**endpoint_params)
print('EndpointArn = {}'.format(endpoint_response['EndpointArn']))
```

```
EndpointArn = arn:aws:sagemaker:ca-central-1:761424745283:endpoint/banking-20
20-03-05-19-50-41
```

We have to wait until the endpoint is active. The `get_waiter()` API will block until it's ready. While you're waiting, head out to the AWS Management Console of SageMaker and familiarize yourself with the *Endpoint configurations* and *Endpoints* sections and see if you can find the parameters we specified above.

This deployment should take less than 10 minutes. It should start by letting you know that the endpoint is currently in a *Creating* status and if everything works, it will output a status of *InService* once complete. We use the `describe_endpoint()` API to get those status.

In [47]:
```
# get the status of the endpoint
response = sagemaker_client.describe_endpoint(EndpointName=endpoint_name)
status = response['EndpointStatus']
print('EndpointStatus = {}'.format(status))

# wait until the status has changed
sagemaker_client.get_waiter('endpoint_in_service').wait(EndpointName=endpoint_
name)

# print the status of the endpoint
endpoint_response = sagemaker_client.describe_endpoint(EndpointName=endpoint_n
ame)
status = endpoint_response['EndpointStatus']
print('Endpoint creation ended with EndpointStatus = {}'.format(status))

if status != 'InService':
    raise Exception('Endpoint creation failed.')
```

```
EndpointStatus = Creating
Endpoint creation ended with EndpointStatus = InService
```

Predicting with our best 2 models

Now that we have an endpoint, it's time to test with our test data. First, we will need to create a new reference to the endpoint as a `RealTimePredictor()`. This is a high level object that abstracts having to call to either manually call the endpoint or use the [boto3 SDK `invoke_endpoint()` function (https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker-runtime.html#SageMakerRuntime.Client.invoke_endpoint)](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker-runtime.html#SageMakerRuntime.Client.invoke_endpoint). Again, this is something that was done automatically by using the `deploy()` API. The object returned from that API was a `RealTimePredictor`.

To create it, we will need to pass the following:

- endpoint name that we defined above
- sagemaker session to be able to make a call to SageMaker to get information about the endpoint
- serializer and content_type set to the same values as we did in Lab 4

In [48]:
```
xgb_endpoint1 = sagemaker.predictor.RealTimePredictor(
    endpoint=endpoint_name,
    sagemaker_session=sess,
    serializer=csv_serializer,
    content_type='text/csv'
)
```

We can now do the same as Lab 4 by splitting the data by 500 rows and doing a prediction. Refer to that lab if you forgot how it works!

In [50]:
```
rows=500
split_test_data_array = np.array_split(test_data_no_prediction, int(test_data_
no_prediction.shape[0] / float(rows) + 1))
predictionsResult = ''
for test_data_500rows in split_test_data_array:
    predictionsResult = ','.join([predictionsResult, xgb_endpoint1.predict(tes
t_data_500rows).decode('utf-8')])

predictions = np.fromstring(predictionsResult[1:], sep=',')
```

We can now look at the confusion matrix the same what as we did in Lab 4 using the `crosstab()` function of pandas.

In [51]:
```
rounded_predictions = np.where(predictions > 0.5, 1, 0)

# Also called a 'confusion matrix'
pd.crosstab(index=test_data['y_yes'],
            columns=rounded_predictions,
            rownames=['truth'], colnames=['predictions'])
```

Out[51]:

predictions	0	1
truth		
0	3592	46
1	380	101

We can also see our precision, recall and F1 score via the same method as Lab 4 using the `precision_recall_fscore_support()` function.

In [52]:
```
score = precision_recall_fscore_support(
            test_data['y_yes'], rounded_predictions, average='binary')

# Precision, recall, F1 score
print(score)
```

```
(0.6870748299319728, 0.20997920997921, 0.321656050955414, None)
```

You can compare those values with what you received in Lab 4. However, those numbers may not be better. The reason for that is that first, you are using the top1 and top2 models. This means that if you rerun the last 3 cells, you may get a different answer each time. You could modify the endpoint to send 100% of the traffic to the top1 variant (model) and see if that gives you better results. You could do that in the AWS Management Console of SageMaker under the *Endpoints* section, under your endpoint itself in its *Endpoint runtime settings* and updating the weights to 1 for top1 and 0 for top2. However, this is an exercise for you to do. Another reason why this would yield to worse predictions is that we only trained 9 models with our automatic model tuning. We may have been lucky the first time we picked hyperparameters and the 3 sequential trainings we did didn't finish optimizing it. So you could run the automatic tuning job for more than 9 models and see if that will give you better results. Again, this is an exercise for another time unless you still have time in this session.

Now that we are done with this lab, we can delete the endpoint, its configuration and the two models. Although, the files will still exist in S3 so you would need to delete everything under the S3 bucket used by SageMaker. We left the deletion commented as some participants prefers to run the entire notebook and may not want to delete all of those. Uncomment the following code and run it.

In [53]:
```
sagemaker_client.delete_endpoint(EndpointName=endpoint_name)
sagemaker_client.delete_endpoint_config(EndpointConfigName=endpoint_config_nam
e)
sagemaker_client.delete_model(ModelName=top1_model_name)
sagemaker_client.delete_model(ModelName=top2_model_name)
print('The bucket used by SageMaker is:', bucket)
```

```
The bucket used by SageMaker is: sagemaker-ca-central-1-761424745283
```

Congratulations! You now know more about Amazon SageMaker.

We've seen how to:

- load and pre-process columnar data with Pandas,
- build training, validation and test sets,
- use the high-level SageMaker SDK,
- train a model with a built-in algorithm,
- predict with a model deployed to an HTTPS endpoint,
- use automatic model tuning to find the best hyperparameters,
- deploy any model on-demand and predict with it.

Now it's your turn to build! We thank you for attending this workshop and we hope you had a good time. Please let your facilitators know if you did or if you want something changed!

Workshop adapted by Jonathan Dion:

- LinkedIn: https://linkedin.com/in/jotdion (https://linkedin.com/in/jotdion)
- Twitter: https://twitter.com/jotdion (https://twitter.com/jotdion)

Workshop originally created by Julien Simon:

- LinkedIn: https://linkedin.com/in/juliensimon (https://linkedin.com/in/juliensimon)
- Twitter: https://twitter.com/julsimon (https://twitter.com/julsimon)
- Medium: https://medium.com/@julsimon (https://medium.com/@julsimon)