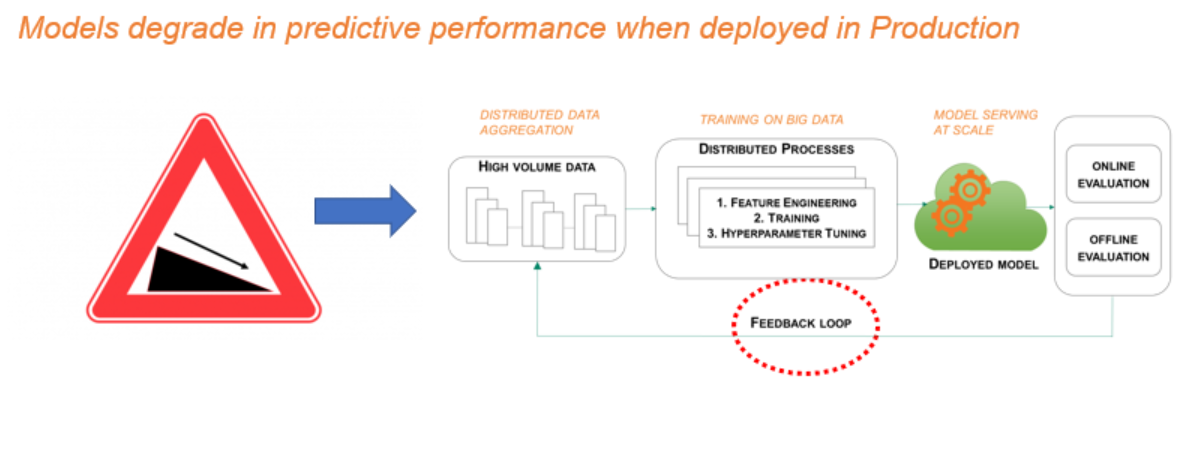
### **CHAPTER 14:** Model Accuracy Degradation and Feedback Loop - 22 pages

**Introduction - Model Accuracy Degradation**

In real world scenarios, the performance of deployed machine learning models degrades over time. To explain, in the case of fraud detection, the models may not capture the evolving fraudulent behavior. Because fraudsters adapt their methods and processes over time to game systems, it is important to retrain fraud detection engines on the latest and greatest data (reflecting anomalous behavior) available. Similarly, in the case of recommender systems, as customer preferences change overtime, it is important for personalization engine to capture these preferences to present the most relevant suggestions to customers.



In this chapter, we will cover:

* Ad Click Conversion – Understand the concept of model performance deterioration using an example on Ad Click Conversion – identify ad clicks that result in mobile app downloads
* What is a feedback loop?
* Retraining model on new data and impact on model performance

**Ad Click Conversion**

Fraud risk is prevalent in almost every industry, from airlines to retail to financial services. Specifically, the risk is high in online advertising. For companies investing in digital marketing, it is important to contain costs from fraudulent clicks on ads. Online advertising can get cost prohibitive if fraudulent behavior is rampant across online ad channels. In this chapter, we will look at ad click data for mobile apps and predict which clicks will likely yield in app downloads. The outcome from this prediction exercise can enable mobile app developers to efficiently allocate online marketing dollars.

Ad click behavior is very dynamic. The behavior changes across time, location and ad channels. A fraudster can write software to automate clicking on mobile app ads and conceal the identity of clicks – clicks can be generated from multiple IP addresses, devices, operating systems, and channels. To catch this dynamic behavior, it is important to retrain classification models to cover new and emerging patterns. Implementing feedback loop becomes critical to be able to accurately determine which clicks will result in app downloads.

**Machine Learning Feedback Loop**

In this section, we will demonstrate how retraining a classification model as new data becomes available will enhance model performance – i.e. better predict which ad clicks will result in mobile app downloads. Although the dataset ([link](https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/data)) contains 200 million clicks across 4 days (Monday through Thursday), we will sample the dataset for each day – select 600k clicks per day.

The dataset contains following elements:

* ip: ip address of click.
* app: app id for marketing.
* device: device type id of user mobile phone (e.g., iphone 6 plus, iphone 7)
* os: os version id of user mobile phone
* channel: channel id of mobile ad publisher
* click\_time: timestamp of click (UTC)
* attributed\_time: if user downloads the app for after clicking an ad, this is the time of the app download
* is\_attributed: the target that is to be predicted, indicating the app was downloaded

Having access to latest and greatest data is a challenge. Data lake and data warehouse environments typically lag by a day (24 hours). When predicting whether clicks that occurred towards end of the day on Thursday will result in app downloads, it is important to have current data up to and including Thursday, excluding the clicks which we are scoring, for model training.

To understand the significance of feedback loop, we will train tree-based model (XGBoost) to predict the probability that an ad click (related to an app) results in app download:

* Experiment 1: Train on click data for Monday and Tuesday and predict/score portion of the clicks from Thursday (clicks from later part of the day)
* Experiment 2: Let’s assume that we have more data available in data lake environment to retrain the classification model. We will train on click data for Monday, Tuesday, & Wednesday and predict/score portion of the clicks from Thursday
* Experiment 3: Similarly, we will train on click data for Monday, Tuesday, Wednesday & part of Thursday and predict/score portion of the clicks from Thursday

With each iteration or experiment, you will see the classification model performance measured by AUC (Area Under Curve) increases. AUC is measured by plotting true positive rate against false positive rate. A random classifier has Area Under Curve of 0.5. For an optimal model, AUC should be closer to 1. In other words, the true positive rate (proportion of the app downloads that you’ve correctly identified) should be higher than false positive rate (proportion of clicks that did not result in app downloads but identified as yielding app downloads).

*Using Amazon EMR (Elastic MapReduce) for data sampling*

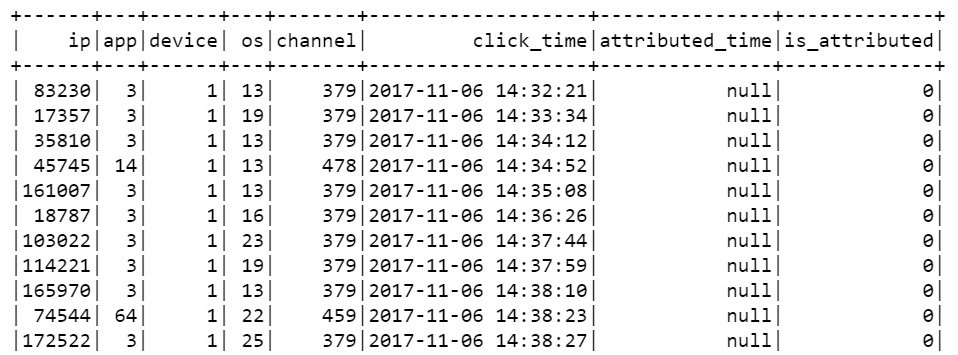
Refer to the earlier chapter *Working with SageMaker* for how to create EMR notebooks. In this section, we will focus on wrangling big data to create the data samples we need. (Create a Spark cluster and run it. A cluster must be running for an EMR notebook to start).

*Read Ad Click Data*

#Read the compressed training dataset containing details on every click

ad\_track\_df = spark.read.option("header","true").option("separator", ",").csv(('s3://' + os.path.join(s3\_bucket, fileName)))

ad\_track\_df.show()



*Partition Ad Click Data by Day*

# Filter sql dataframe by date - day 1, 2, 3, 4, and 5

ad\_track\_df = ad\_track\_df.select("\*", F.col("click\_time").cast("date").alias("click\_date"))

days = ['2017-11-06', '2017-11-07', '2017-11-08', '2017-11-09']

# Mon, # Tues, # Wed, # Thurs

dict\_of\_df = {}

for idx, dy in enumerate(days):

key\_name = 'ad\_track\_day' + str(idx+1)

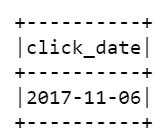
dict\_of\_df[key\_name] = ad\_track\_df.filter(ad\_track\_df.click\_date == dy)

dict\_of\_df.keys()



# Get unique value of click\_date column

dict\_of\_df['ad\_track\_day1'].select('click\_date').distinct().show()



Select sample size for each of the days

#select subset of rows

num\_rows = 600000

for k in dict\_of\_df.keys():

dict\_of\_df[k] = dict\_of\_df[k].limit(num\_rows)

#view the distribution of is\_attributed

for k in dict\_of\_df.keys():

dict\_of\_df[k].groupBy(F.col('is\_attributed')).count().show()

for k in dict\_of\_df.keys():

dict\_of\_df[k] = dict\_of\_df[k].limit(num\_rows)

|  |  |  |  |
| --- | --- | --- | --- |
| Monday | Tuesday | Wednesday | Thursday |
|  |  |  |  |

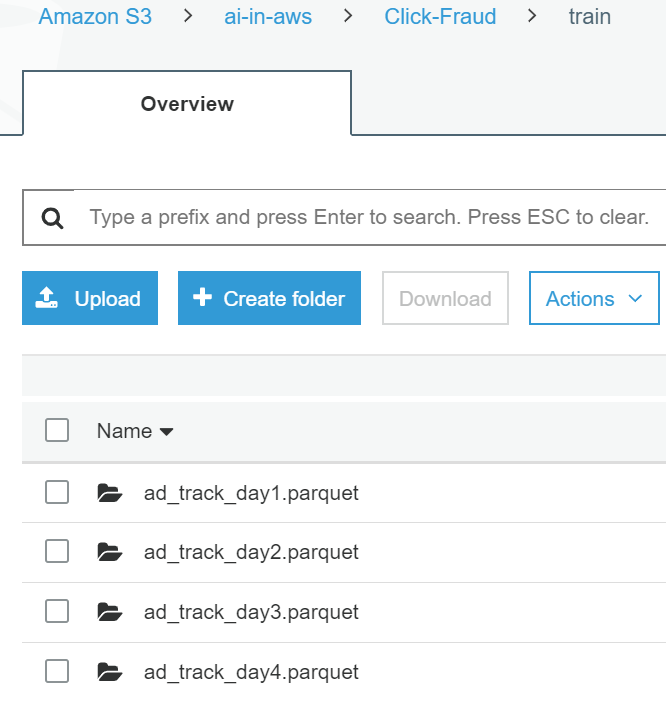
Data is heavily imbalanced - 0.1% of clicks result in app downloads

# Write the data in parquet format and save to S3 bucket

for k in dict\_of\_df.keys():

dict\_of\_df[k].write.parquet('s3://' + s3\_bucket + '/' + train\_prefix + "/" + k + ".parquet", mode='overwrite')

The parquet files for each of the days are uploaded to designated S3 bucket.



*Using Amazon SageMaker XGBoost algorithm to classify ad click data*

Amazon SageMaker offers built in tools and capabilities to create machine learning pipelines that incorporate feedback loop. Since machine learning pipelines are covered in chapter *Create Machine Learning Inference Pipelines with Amazon SageMaker,* we will focus on the significance of incorporating feedback loop.

#Install relevant python packages

!pip install --upgrade pip

!pip install pyarrow

!pip install joblib

!pip install xgboost

Read the dataset from S3 bucket

s3\_bucket = 'ai-in-aws'

s3\_prefix = 'Click-Fraud'

s3\_train\_prefix = os.path.join(s3\_prefix, 'train')

s3\_val\_prefix = os.path.join(s3\_prefix, 'val')

s3\_output\_prefix = os.path.join(s3\_prefix, 'output')

s3\_train\_fn = 'train\_sample.csv.zip'

# Location of dataset on local ec2 instance where SageMaker is running

sm\_train\_loc = 'click\_fraud/train/'

sm\_val\_loc = 'click\_fraud/val/'

#Read the prepared dataset from s3 bucket

s3 = s3fs.S3FileSystem()

file\_name = 'ad\_track\_day'

fn\_ext = '.parquet'

num\_days = 4

dict\_of\_ad\_trk\_df = {}

for i in range(1, num\_days+1):

dataset\_name = 's3://' + os.path.join(s3\_bucket, s3\_train\_prefix, file\_name+str(i)+fn\_ext)

dict\_of\_ad\_trk\_df[file\_name+str(i)] = pq.ParquetDataset(dataset\_name, filesystem=s3).read\_pandas().to\_pandas()

Now, let’s look at the functions we’ve defined to explore data, create train, validation & test datasets and evaluate model performance.

run clickfraud\_processing.py

Once we create train, validation and test datasets, we’ll use *upload\_to\_s3* to uploaded them to relevant bucket (and folders) on S3

def upload\_to\_s3(bucket, channel, file):

"""

input: S3 bucket name, folder on the bucket, RecordIO file to upload

"""

s3 = boto3.resource('s3')

data = open(file, "rb") # read in binary mode

key = channel

#Key is location on S3 bucket, Body - binary data

s3.Bucket(bucket).put\_object(Key=key, Body=data)

From the click\_time column, we’ll create date related features – day, hour, minute and second. These features will help uncover if there are certain click patterns given day of the week and hour of the day.

def create\_date\_ftrs(df\_ckFraud, col\_name):

"""

create day, hour, minute, second features

"""

df\_ckFraud = df\_ckFraud.copy()

df\_ckFraud['day'] = df\_ckFraud[col\_name].dt.day.astype('uint8') ## dt is accessor object for date like properties

df\_ckFraud['hour'] = df\_ckFraud[col\_name].dt.hour.astype('uint8')

df\_ckFraud['minute'] = df\_ckFraud[col\_name].dt.minute.astype('uint8')

df\_ckFraud['second'] = df\_ckFraud[col\_name].dt.second.astype('uint8')

return df\_ckFraud

For each of the ad click related characteristics, such as type of app, device, operating system, and IP address the click is originating from, it is important to not only know their individual frequency but also their combination frequency. Perhaps app downloads happen when ad clicks are for certain app and/or when clicks come on certain day of week and from certain device.

def count\_clicks(df\_ckFraud):

# Get all ip address related click counts

ip\_wday\_cnt = df\_ckFraud.groupby(['ip', 'day', 'hour'])['os'].count().reset\_index()

ip\_wday\_cnt.columns = ['ip', 'day', 'hour', 'clicks\_by\_ip\_day\_hr']

ip\_channel\_cnt = df\_ckFraud.groupby(['ip', 'hour', 'channel'])['os'].count().reset\_index()

ip\_channel\_cnt.columns = ['ip', 'hour', 'channel', 'clicks\_by\_ip\_hr\_chnl']

ip\_os\_cnt = df\_ckFraud.groupby(['ip', 'hour', 'os'])['channel'].count().reset\_index()

ip\_os\_cnt.columns = ['ip', 'hour', 'os', 'clicks\_by\_ip\_hr\_os']

ip\_app\_cnt = df\_ckFraud.groupby(['ip', 'hour', 'app'])['os'].count().reset\_index()

ip\_app\_cnt.columns = ['ip', 'hour', 'app', 'clicks\_by\_ip\_hr\_app']

ip\_device\_cnt = df\_ckFraud.groupby(['ip', 'hour', 'device'])['os'].count().reset\_index()

ip\_device\_cnt.columns = ['ip', 'hour', 'device', 'clicks\_by\_ip\_hr\_device']

df\_ckFraud = pd.merge(df\_ckFraud, ip\_wday\_cnt, on=['ip', 'day', 'hour'], how='left', sort=False)

df\_ckFraud = pd.merge(df\_ckFraud, ip\_channel\_cnt, on=['ip', 'hour', 'channel'], how='left', sort=False)

df\_ckFraud = pd.merge(df\_ckFraud, ip\_os\_cnt, on=['ip', 'hour', 'os'], how='left', sort=False)

df\_ckFraud = pd.merge(df\_ckFraud, ip\_app\_cnt, on=['ip', 'hour', 'app'], how='left', sort=False)

df\_ckFraud = pd.merge(df\_ckFraud, ip\_device\_cnt, on=['ip', 'hour', 'device'], how='left', sort=False)

return df\_ckFraud

The unique ids of each of the categorical columns – app, device, os, channel – are not useful in and of itself. For a tree-based model, for example, a lower app id is not better than higher app id or vice versa in terms of predicting app downloads. We will therefore calculate frequency of each of these categorical variables.

def encode\_cat\_ftrs(df\_ckFraud):

cat\_ftrs = ['app','device','os','channel']

for c in cat\_ftrs:

df\_ckFraud[c+'\_freq'] = df\_ckFraud[c].map(df\_ckFraud.groupby(c).size() / df\_ckFraud.shape[0])

# indexer = pd.factorize(df\_ckFraud[c], sort=True)[1]

# df\_ckFraud[c+'\_factrzr'] = indexer.get\_indexer(df\_ckFraud[c])

return df\_ckFraud

For each of the three experiments that we’re going to run, we will evaluate how the importance of features changes.

def plot\_ftr\_imp(model\_file):

tar = tarfile.open(model\_file, "r:gz")

for member in tar.getmembers():

f = tar.extractfile(member)

if f is not None:

content = f.read()

tar.extractall()

fil = open('xgboost-model', 'rb')

## Load model file back into generate predictions & view feature importance

xgb\_local\_ckFraud = joblib.load(fil)

## Close

fil.close()

## Chart variable importance

fig, ax = plt.subplots(figsize=(6,6))

xgbst.plot\_importance(xgb\_local\_ckFraud, max\_num\_features=8, height=0.8, ax=ax, show\_values = False)

plt.title('Click Fraud Model Feature Importance')

plt.show()

To better create features, we will need to understand how the most frequently (top 10, let’s say) occurring predictors (or categorical variables) differ when app is downloaded vs not downloaded

def plot\_clickcnt\_ftr(df\_ckFraud, ftr, exp\_num):

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

df\_ckFraud.groupby(ftr)['day'].count().sort\_values(ascending=False).head(10).plot(kind='bar', ax=ax1)

ax1.set\_xlabel(ftr)

ax1.set\_ylabel("Number of Clicks")

ax1.set\_title("Experiment #{}: Number of Clicks by {}".format(exp\_num, ftr))

df\_ckFraud[df\_ckFraud['is\_attributed'] == 1].groupby(ftr)['day'].count().sort\_values(ascending=False).head(10).plot(kind='bar', ax=ax2)

ax2.set\_xlabel(ftr)

ax2.set\_ylabel("Number of Clicks")

ax2.set\_title(("Experiment #{}: Number of Clicks by {} when apps are downloaded").format(exp\_num, ftr))

fig.subplots\_adjust(wspace=1)

*Prepare Data for Experiments*

Create chunks of data for each of the experiments and assign appropriate data type for each of the columns

df\_ckFraud\_exp1 = pd.concat([dict\_of\_ad\_trk\_df[key] for key in ["ad\_track\_day1", "ad\_track\_day2"]], ignore\_index=True)

df\_ckFraud\_exp2 = pd.concat([dict\_of\_ad\_trk\_df[key] for key in ["ad\_track\_day1", "ad\_track\_day2", "ad\_track\_day3"]], ignore\_index=True)

df\_ckFraud\_exp3 = pd.concat([dict\_of\_ad\_trk\_df[key] for key in ["ad\_track\_day1", "ad\_track\_day2", "ad\_track\_day3", "ad\_track\_day4"]], ignore\_index=True)

# Set the data types for each of the columns

# Ensure we have appropriate column types

cat\_var = ['ip','app','device','os','channel', 'is\_attributed']

for var in cat\_var:

df\_ckFraud\_exp1[var] = df\_ckFraud\_exp1[var].astype('uint32')

df\_ckFraud\_exp2[var] = df\_ckFraud\_exp2[var].astype('uint32')

df\_ckFraud\_exp3[var] = df\_ckFraud\_exp3[var].astype('uint32')

*Explore Data*

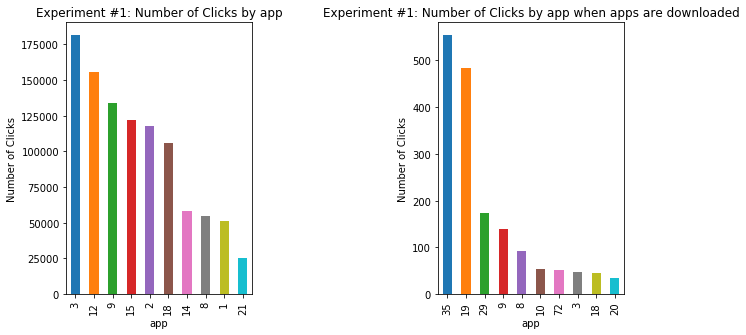
The main goal of exploratory data analysis is to understand if the most frequently occurring factors -- as type of app, device, channel, operating system, and IP address the click is originating from -- result in app downloads.

Experiment 1 – Top 20 apps from Monday and Tuesday

The popular apps, defined by the number of ad clicks pertaining to the app, are not same between when an app is not downloaded vs when it is downloaded. In other words, although certain mobile app ads are clicked widely, they are not necessarily the ones getting downloaded

%matplotlib inline

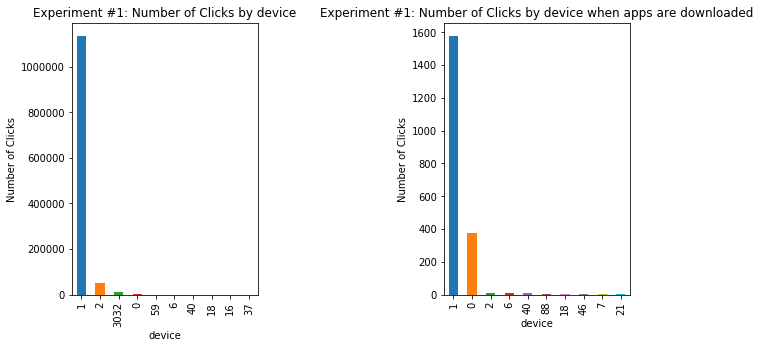
plot\_clickcnt\_ftr(df\_ckFraud\_exp1, 'app', '1')



Top 20 devices from Monday and Tuesday

%matplotlib inline

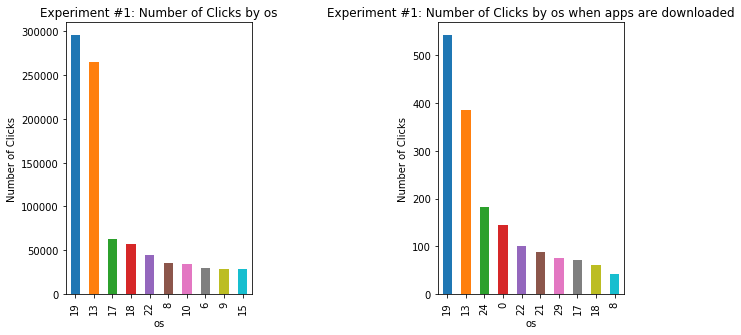
plot\_clickcnt\_ftr(df\_ckFraud\_exp1, 'device', '1')



Top 20 operating system from Monday and Tuesday

%matplotlib inline

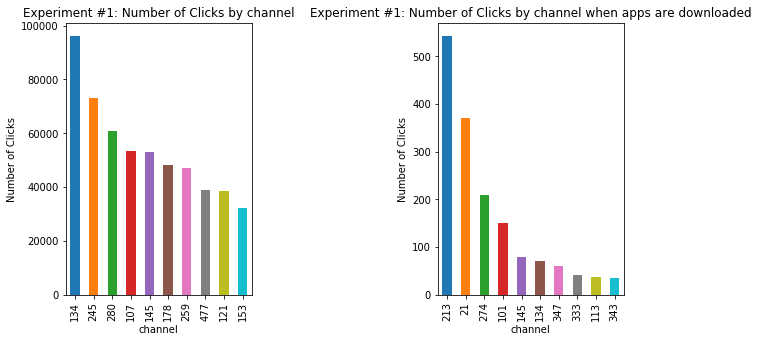
plot\_clickcnt\_ftr(df\_ckFraud\_exp1, 'os', '1')



Top 10 channels from Monday and Tuesday

%matplotlib inline

plot\_clickcnt\_ftr(df\_ckFraud\_exp1, 'channel', '1')



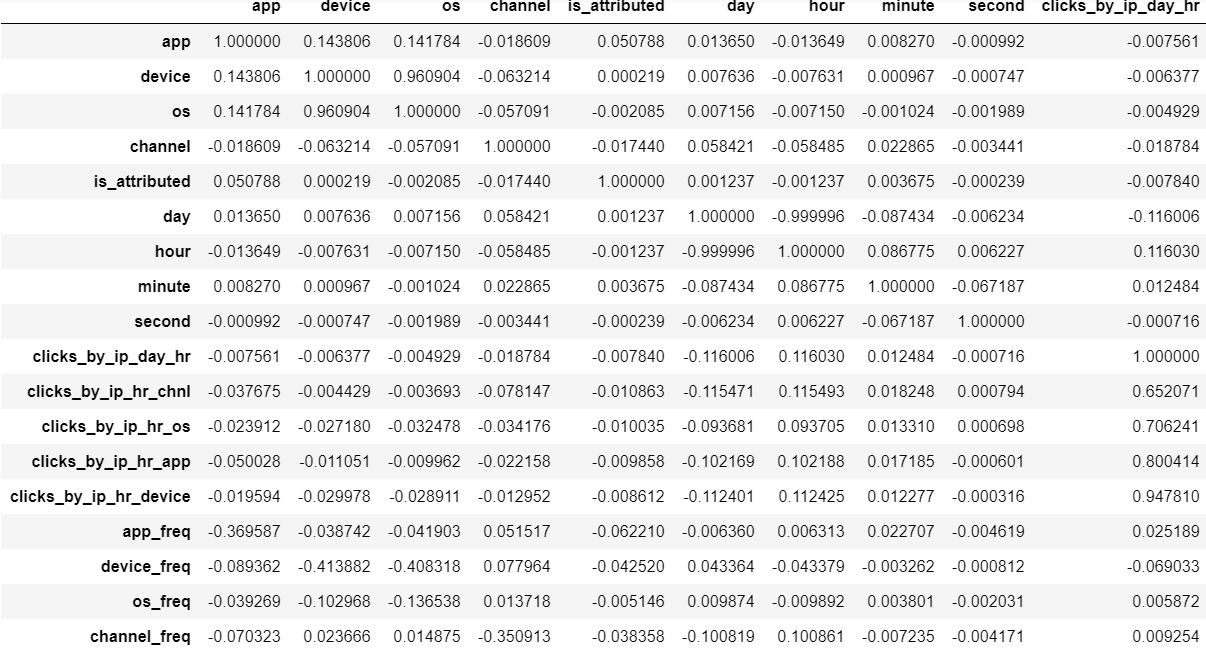
From the above visualization, it is clear that clicks related to certain apps, devices, and channel result in app downloads. In these instances, the popular apps/devices/channels are not largely shared between when clicks result in app downloads vs not.

The correlation matrix below also confirms the above hypothesis.

# Correlation

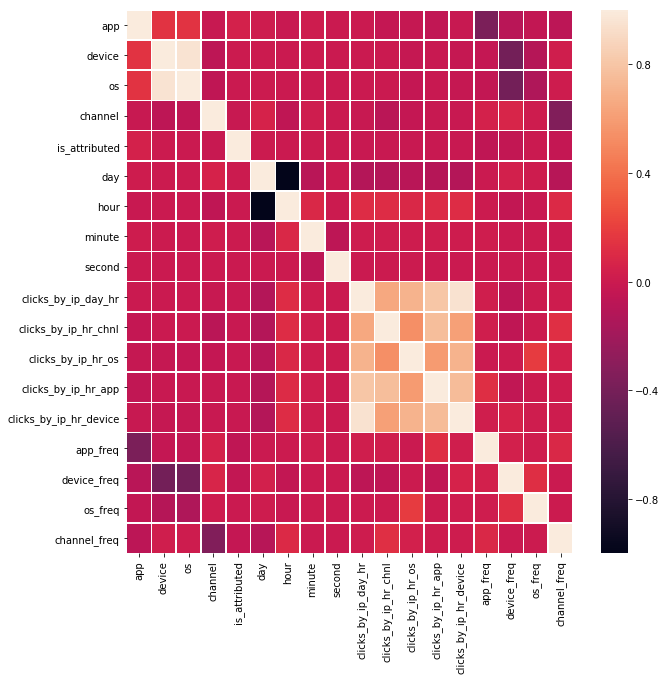
df\_ckFraud\_exp1.corr()

Partial correlation matrix is displayed



fig, ax = plt.subplots(figsize=(10, 10))

sns.heatmap(df\_ckFraud\_exp1.corr(), linewidth=.5, ax=ax)

*Create Features*

Create time features

date\_var = ['click\_time', 'click\_date', 'attributed\_time']

col\_name = 'click\_time'

for var in date\_var:

df\_ckFraud\_exp1[var] = pd.to\_datetime(df\_ckFraud\_exp1[var])

df\_ckFraud\_exp2[var] = pd.to\_datetime(df\_ckFraud\_exp2[var])

df\_ckFraud\_exp3[var] = pd.to\_datetime(df\_ckFraud\_exp3[var])

df\_ckFraud\_exp1 = create\_date\_ftrs(df\_ckFraud\_exp1, col\_name)

df\_ckFraud\_exp2 = create\_date\_ftrs(df\_ckFraud\_exp2, col\_name)

df\_ckFraud\_exp3 = create\_date\_ftrs(df\_ckFraud\_exp3, col\_name)

Create features reflecting frequency of occurrence of each of the categorical variables

# Create appropriate features -- from a given ip address and time, number of clicks generated

df\_ckFraud\_exp1 = count\_clicks(df\_ckFraud\_exp1)

df\_ckFraud\_exp2 = count\_clicks(df\_ckFraud\_exp2)

df\_ckFraud\_exp3 = count\_clicks(df\_ckFraud\_exp3)

# Drop click time, attribution time and ip

df\_ckFraud\_exp1 = df\_ckFraud\_exp1.drop(['click\_time', 'attributed\_time', 'click\_date', 'ip'], axis=1)

df\_ckFraud\_exp2 = df\_ckFraud\_exp2.drop(['click\_time', 'attributed\_time', 'click\_date', 'ip'], axis=1)

df\_ckFraud\_exp3 = df\_ckFraud\_exp3.drop(['click\_time', 'attributed\_time', 'click\_date', 'ip'], axis=1)

Encode categorical columns -- frequency encoding

df\_ckFraud\_exp1 = encode\_cat\_ftrs(df\_ckFraud\_exp1)

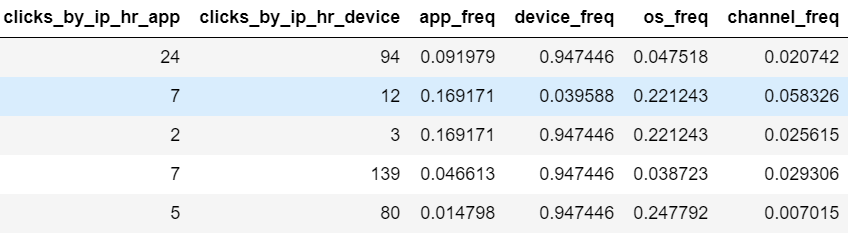
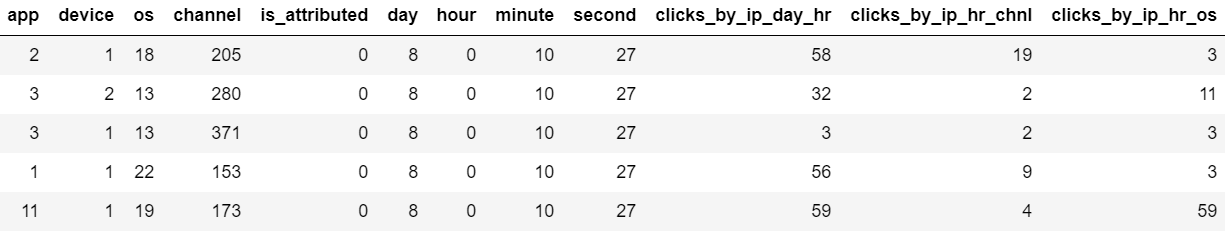
df\_ckFraud\_exp2 = encode\_cat\_ftrs(df\_ckFraud\_exp2)

df\_ckFraud\_exp3 = encode\_cat\_ftrs(df\_ckFraud\_exp3)

Check the prepared dataset

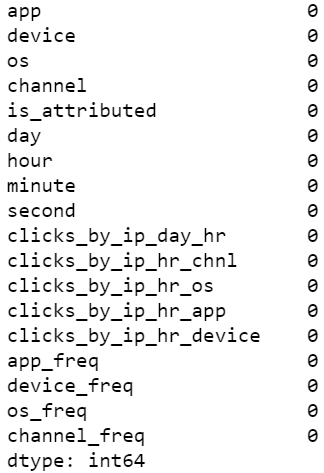
# Check the prepared dataset

df\_ckFraud\_exp2.tail()



Ensure there are no missing values

df\_ckFraud\_exp3.apply(lambda x: sum(x.isnull()))



*Run Experiments*

To understand the significance of feedback loop, we will train tree-based model (XGBoost) to predict the probability that an ad click (related to an app) results in app download:

* Experiment 1: Train on click data for Monday and Tuesday and predict/score portion of the clicks from Thursday (clicks from later part of the day)
* Experiment 2: Let’s assume that we have more data available in data lake environment to retrain the classification model. We will train on click data for Monday, Tuesday, & Wednesday and predict/score portion of the clicks from Thursday
* Experiment 3: Similarly, we will train on click data for Monday, Tuesday, Wednesday & part of Thursday and predict/score portion of the clicks from Thursday

Set current experiment: Fit the classification model for each of the experiments, host the trained model and run inferences.

current\_experiment = df\_ckFraud\_exp3

exp\_prefix = 'exp3'

Create training, validation and test datasets

print(s3\_train\_prefix + '/' + exp\_prefix + '/train\_ckFraud.csv')

*Test Dataset*

We will select last 5% of clicks from Thursday as test dataset. Also, we will create new dataset for experiment 3, excluding the observations from test dataset.

# Create test dataset

# Sort by hour, minute and second --> pick the last 5% (120,000 records)

test\_data = df\_ckFraud\_exp3.sort\_values(['day', 'hour', 'minute', 'second'], ascending=False).head(n=120000)

# Rearrange test data so that is\_attributed is the first column

test\_data = pd.concat([test\_data['is\_attributed'], test\_data.drop(['is\_attributed'], axis=1)], axis=1)

# Create new dataset for experiment 3 (this excludes data from test dataset)

df\_ckFraud\_fil\_exp3 = df\_ckFraud\_exp3.sort\_values(['day', 'hour', 'minute', 'second'], ascending=True).head(n=2280000)

When running experiment 3, set the current experiment to newly created experiment #3 dataset

if exp\_prefix=='exp3':

current\_experiment = df\_ckFraud\_fil\_exp3

Edit the dataset to ensure the first column contains the label (target) – the format in which XGBoost requires training data.

# Create dataset in a format required by XGBoost

current\_experiment = pd.concat([current\_experiment['is\_attributed'], current\_experiment.drop(['is\_attributed'], axis=1)], axis=1)

*Train and validation Dataset*

Create train (80%) and validation (20%) datasets

train\_data, validation\_data = np.split(current\_experiment.sample(frac=1, random\_state=4567), [int(0.7 \* len(current\_experiment))])

# Save training and validation csv to local sagemaker instance

train\_data.to\_csv(sm\_train\_loc + 'train\_ckFraud.csv', header=False, index=False)

validation\_data.to\_csv(sm\_val\_loc + 'val\_ckFraud.csv', header=False, index=False)

# Upload csv to S3 bucket

upload\_to\_s3(s3\_bucket, s3\_train\_prefix + '/' + exp\_prefix + '/train\_ckFraud.csv', sm\_train\_loc + 'train\_ckFraud.csv')

upload\_to\_s3(s3\_bucket, s3\_val\_prefix + '/' + exp\_prefix + '/val\_ckFraud.csv', sm\_val\_loc + 'val\_ckFraud.csv')

*Prepare for Training*

Define the location of train and validation datasets

s3\_input\_train = sagemaker.s3\_input(s3\_data='s3://{}/{}/train/{}/{}'.format(s3\_bucket, s3\_prefix, exp\_prefix, 'train\_ckFraud.csv'), content\_type='csv')

s3\_input\_validation = sagemaker.s3\_input(s3\_data='s3://{}/{}/val/{}/{}'.format(s3\_bucket, s3\_prefix, exp\_prefix, 'val\_ckFraud.csv'), content\_type='csv')

Get SageMaker XGBoost Training Image

sess = sagemaker.Session()

role = get\_execution\_role()

#Get docker image for the XGBoost algorithm

container = get\_image\_uri(boto3.Session().region\_name, 'xgboost') #input: region name, name of the algorithm

As we have seen above, the dataset is highly imbalanced. To account for it, we will use one of the hyperparameters – scale\_pos\_weight – to give clicks that resulted in app downloads more weight. These clicks are heavily underrepresented in the dataset.

Scale\_pos\_weight is

[1 – (Number of positive observations) / (Number of total observations)] \* 100

# Unbalanced data

num\_positives = current\_experiment.groupby('is\_attributed').size()[1]

num\_negatives = current\_experiment.groupby('is\_attributed').size()[0]

scale\_pos\_weight = 100 - (num\_positives / (num\_positives + num\_negatives)) \* 100 # \*\*number of positive samples\*\* / \*\*total samples\*\*

scale\_pos\_weight



*Training*

For training XGBoost model, the following hyperparameters are defined (only a few are reported). For detailed information, refer to [aws docs](https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost_hyperparameters.html).

* Eta – learning rate
* Gamma - node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split
* Subsample – fraction of the observations to be randomly sampled for each tree
* Colsample\_bytree: fraction of Columns to be randomly sampled for each tree
* Scale\_pos\_weight: Defined above
* Alpha – L1 regularization
* Lambda – L2 regularization

xgb = sagemaker.estimator.Estimator(container,

role,

train\_instance\_count=1,

train\_instance\_type='ml.m4.xlarge',

output\_path='s3://{}/{}/output/{}'.format(s3\_bucket, s3\_prefix, exp\_prefix),

sagemaker\_session=sess)

xgb.set\_hyperparameters(max\_depth=4,

eta=0.3,

gamma=0,

min\_child\_weight=6, #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

colsample\_bylevel = 0.8,

colsample\_bytree = 0.8,

subsample=0.8,

silent=0,

scale\_pos\_weight=scale\_pos\_weight,

objective='binary:logistic',

num\_round=100)

xgb.fit({'train': s3\_input\_train, 'validation': s3\_input\_validation})

*Download the trained model from s3 bucket to plot feature importance*

# Download the trained model from S3 bucket

s3\_output\_prefix = 'output'

s3\_output\_fn = 'model.tar.gz'

sm\_output\_loc = 'click\_fraud/output/'

#dwnld\_op\_fn = sm\_output\_loc + s3\_output\_fn

#Download model from s3 bucket

data\_loc='s3://{}/{}/{}/{}'.format(s3\_bucket, s3\_prefix, s3\_output\_prefix, exp\_prefix)

!aws s3 cp $data\_loc $sm\_output\_loc$exp\_prefix --exclude "\*" --include "\*.tar.gz" --recursive

## Unpack model file

exp\_lst = ['exp1', 'exp2', 'exp3']

for exp in exp\_lst:

model\_file = os.path.join(sm\_output\_loc, exp, s3\_output\_fn)

plot\_ftr\_imp(model\_file)

After all the three experiments are run, we plot feature importance. Most of the key predictors maintained their importance as more data became available (Mon+Tues; Mon+Tues+Wed; Mon+Tues+Wed+Thurs)

|  |  |  |
| --- | --- | --- |
| Experiment 1 | Experiment 2 | Experiment 3 |
|  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| f0 | f1 | f2 | f3 | | f4 | f5 | f6 | f7 | f8 | f9 | f10 | f11 | f12 | f13 | f14 | f15 | f16 |
|  | | | |  | | | | | | | | | | | | | | |

*Evaluate Model Performance*

We will now evaluate performance across all the three experiments – let’s deploy all the three trained models as end points.

# predict whether the click will result in app download

# deploy trained model in SageMaker

# Use SageMaker Python SDK

model\_loc = os.path.join(data\_loc, s3\_output\_fn)

xgb\_model = Model(model\_data=model\_loc, image=container, role=role)

xgb\_model.deploy(initial\_instance\_count=1,

instance\_type='ml.m4.xlarge')

*Measure performance Across All Trained Models*

After the endpoints are available, define endpoint list below:

# Get the deployed endpoint names

enpoint\_lst = ['xgboost-2019-05-09-19-20-56-069' , 'xgboost-2019-05-09-18-53-55-807', 'xgboost-2019-05-09-19-50-06-235'] #exp1, exp2, exp3

predictions = {}

for ind, endpoint in enumerate(enpoint\_lst):

xgb\_predictor = sagemaker.predictor.RealTimePredictor(endpoint, sagemaker\_session=sess, serializer=csv\_serializer, deserializer=None, content\_type='text/csv', accept=None)

predictions[exp\_lst[ind]] = xgb\_predictor.predict(test\_data.as\_matrix()[:10000, 1:]).decode('utf-8')

predictions[exp\_lst[ind]] = np.fromstring(predictions[exp\_lst[ind]][1:], sep=',') #A new 1-D array initialized from text data in a string.

# pd.crosstab(index=test\_data.as\_matrix()[:10000, 0], columns=np.round(predictions[exp\_lst[ind]]), rownames=['actual'], colnames=['predictions'])

# plt.hist(predictions1)

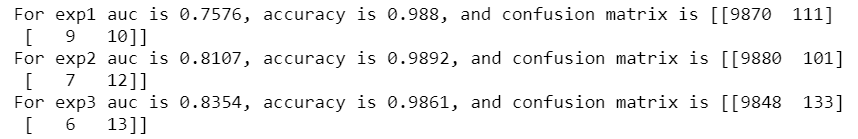
# plt.show()

c\_mat = confusion\_matrix(test\_data.as\_matrix()[:10000, 0], np.round(predictions[exp\_lst[ind]])) ## Predicted vs. actual outcome

accuracy = round(accuracy\_score(test\_data.as\_matrix()[:10000, 0], np.round(predictions[exp\_lst[ind]])), 4) ## Overall accuracy

auc = round(roc\_auc\_score(test\_data.as\_matrix()[:10000, 0], np.round(predictions[exp\_lst[ind]])), 4)

print('For {} auc is {}, accuracy is {}, and confusion matrix is {}'.format(exp\_lst[ind], auc, accuracy, c\_mat))

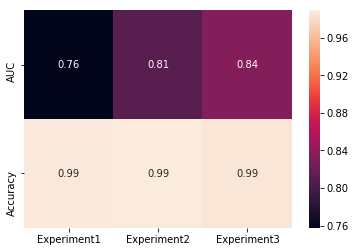


Index= ['AUC', 'Accuracy']

df\_model\_perf = pd.DataFrame({'Experiment1': [0.7576, 0.988], 'Experiment2': [0.8107, 0.9892], 'Experiment3': [0.8354, 0.9861]}, index=Index)

sns.heatmap(df\_model\_perf, annot=True)

The model performance (Area Under Curve) improved as more data on click behavior became available.



*Conclusion*

Implementing feedback loop in machine learning lifecycle is critical to maintaining and enhancing model performance, adequately addressing business objectives – whether it is fraud detection or capturing user preferences for recommendations. With Amazon SageMaker built-in algorithms and other AWS services, such as EMR notebooks (Spark Cluster), it is seamless to maintain and enhance performance of machine learning models.

**References**

<https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/data>

<https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost_hyperparameters.html>