Building a Simple Prediction Model with Unified Leads and Transactions Data

BA Basu, Abhi (CAI - Austin)
Principal Technical Architect

During my prior work with Unified Leads and Transactions (this blog describes how to get access to these datasets - https://coxautoinc.sharepoint.com/sites/IMS-Architecture/SitePages/My-Data-Analytics-Journey-at-Cox-Automotive.aspx) I was able to obtain and curate these two datasets. I wanted to show an example of how to build a simple prediction model with these datasets, with full understanding that this dataset may not yield very accurate results.

Refining the Dataset

Thanks to assistance from the Enterprise Data/Enterprise Data Foundations team, I was able to use Snowflake as my data warehouse. The fast SQL query capabilities and storage allowed me to build temporary tables that held 6 months (July-December, 2019) of Leads and Transactions data, instead of 3 months of data that I was using in the above mentioned blog (due to Redshift access issues and local data processing limitations). Allowing Snowflake to do all the heavy lifting in terms of data exploration, some aggregations, joining tables and building staging tables accelerated my work significantly.

Building the Tables in Snowflake

```
## TRANSACTION RETAIL TABLE
create table CRS.ABHI_CRS.UNIFIED_TRANS_RETAIL_2019 as
select t1.UNFD_TXN_ALL_KEY, t1.VIN, t1.SALE_PRICE, t1.MILEAGE, t1.CPO_FLG, t1.VEHICLE_TYPE,
t1.TXN_LOCATION_ZIP_CD, t1.TXN_LOCATION_NM, t1.TIME_ON_LOT, t1.SOLD_DT,
t1.SALE_TYPE,t1.DATA_SRC, t1.COUNTRY, t1.CONDITION_VALUE, t1.CA_VEHICLE_YEAR, t1.CA_VEHICLE_TRIM,
t1.CA_VEHICLE_BODY_STYLE, t1.CA_CLIENT_ZIP_CD_SELLER,
t1.CA_CLIENT_NAME_SELLER, t1.CA_CLIENT_ID_SELLER, t1.BUYER_ZIP_CD,
t2.SRC_COLOR_EXT, t2.SRC_COLOR_INT, t2.SRC_DRIVETRAIN, t2.SRC_ENGINE, t2.SRC_MAKE, t2.SRC_MODEL, t2.SRC_TRANSMISSION,
t2.SRC_TRIM, t2.SRC_YEAR, t2.DATA_SRC_SUB_SRC, t2.MONTHLY_PAYMENT_AMT, t2.TERM, t2.APR, t2.AMOUNT_FINANCED,
t2.FINANCE_TYPE, t2.FRONT_GROSS, t2.BACK_GROSS, t2.MSRP
from  EDPDB.TEST_SCHEMA.TEMP_UNIFIED_TRANSACTIONS_BTG_10FEB20 t1
inner join EDPDB.TEST_SCHEMA.TEMP_UNIFIED_TRANSACTIONSEXT_BTG_10FEB20 t2
on t1.unfd_txn_all_key = t2.unfd_txn_all_key
where
t1.sold_dt like ('2019%')
```

Skip to main content

The Unified Leads extension table was not available to me in Snowflake, so I used the only Leads table.

```
CREATE TABLE CRS.ABHI_CRS.UNIFIED_LEADS_ALL_2019 as
select * from EDPDB.TEST_SCHEMA.TEMP_UNIFIED_LEADSALL_BTG_03FEB20
where lst_updt_dt like ('2019-%')
AND (VIN regexp '[A-HJ-NPR-Z0-9]{17}')
```

```
create table CRS.ABHI_CRS.UNIFIED_LEADS_TRANS_ALL_RETAIL_2019 as
select t1.UNFD_LD_ALL_KEY, t1.UNFD_SHPR_ALL_KEY, t1.VIN,
t1.DATA_SRC, t1.INVTY_TYPE, t1.LD_SRC,
t1.CRNT_LD_STS, t1. CRNT_LD_STS_TYPE, t1.LD_TYPE, t1.LD_MEDIA_TYPE, t1.ZIP_CD,
t1.FIRST_CNTC_DT, t1.LST_UPDT_DT,
t2.MILEAGE, t2.SALE_PRICE, t2.CPO_FLG, t2.VEHICLE_TYPE, t2.TXN_LOCATION_ZIP_CD,
t2.TIME_ON_LOT, t2.SOLD_DT, t2.SALE_TYPE, t2.COUNTRY,
t2.CA_VEHICLE_TRIM, t2.CA_VEHICLE_BODY_STYLE,
t2.SRC_COLOR_EXT, t2.SRC_COLOR_INT, t2.SRC_DRIVETRAIN, t2.SRC_TRIM, t2.SRC_TRANSMISSION, t2.SRC_MODEL, t2.SRC_MAKE,
t2.SRC_ENGINE, t2.SRC_YEAR, t2.CA_CLIENT_ZIP_CD_SELLER, t2.CA_CLIENT_NAME_SELLER, t2.BUYER_ZIP_CD, t2.DATA_SRC_SUB_SRC,
t2.MONTHLY_PAYMENT_AMT, t2.FRONT_GROSS, t2.FINANCE_TYPE, t2.AMOUNT_FINANCED, t2.APR, t2.TERM, t2.BACK_GROSS, t2.MSRP
from
ABHI_CRS.UNIFIED_LEADS_ALL_2019 t1
inner join
CRS.ABHI_CRS.UNIFIED_TRANS_RETAIL_2019 t2
on
(t1.vin = t2.vin)
```

The above table was filtered by date range of July - August, 2019, and written to the final table (CRS.ABHI_CRS.UNIFIED_LEADS_TRANS_ALL_RETAIL_07_12_2019) to be used for the data analysis. The data was saved as a csv file and used in the next step for local analysis using Jupyter Notebook and Python language and libraries.

Making Connection to Snowflake

I have installed Anaconda Python distribution (https://www.anaconda.com/distribution/) which provides base Python (3.x) bundled with numerous useful scientific and data analyses libraries. Anaconda also bundles Jupyter Notebook (https://jupyter.org/) which can be run easily from command line once Anaconda is installed on your system. One big advantage of using Jupyter Notebook is the ability to share the code and transformations with other team members and collaborate in-line. Python3 provides libraries to connect to all types of data storage systems.

Here is an example of code I used to extract a dataset onto my local system using Jupyter Notebook, Python 3 and Snowflake Connector (https://docs.snowflake.com/en/user-guide/python-connector.html).

Skip to main content

In [1]:
```
table_name = 'CRS.ABHI_CRS.UNIFIED_LEADS_TRANS_ALL_RETAIL_07_12_2019'

import snowflake.connector
import pandas as pd
import keyring
import numpy as np
```

In [ ]:
```
# Gets the version
con = snowflake.connector.connect(
  user='abasu',
  password=keyring.get_password("xxxxxx", "xxxxxx"),
  account='cai_labs.us-east-1',
  warehouse='CRS',
  database='EDPDB',
  schema='TEST_SCHEMA',
  role='CRS'
)
```

Note: The Keyring python library is used above as an safe way to store passwords (https://pypi.org/project/keyring/).

Large datasets can be read using chunks in Python like so.

In [ ]:
```
# Create empty list
dfl = []

#Get only data where sale_price was part of the Transaction
sql = """select * from CRS.ABHI_CRS.UNIFIED_LEADS_TRANS_ALL_RETAIL_07_12_2019;"""

# Create empty dataframe
dfs = pd.DataFrame()

i= 0
# Start Chunking
for chunk in pd.read_sql(sql, con, chunksize=500000):

    # Start Appending Data Chunks from SQL Result set into List
    dfl.append(chunk)
    i = i + 1
    print ("Saved chunk " + str(i))

# Start appending data from list to dataframe
dfs = pd.concat(dfl, ignore_index=True)

dfs.sample(10)
```

Extracting the Dataset Locally

Write the dataset locally. This file on disk is around 7.7 GB in size and consists of 23.3 million records.

Skip to main content

Model Selection

Given we have one binary dependent variable (y) with only two possible outcomes - Sold or Not Sold, we are going to build a Logistic Regression Model. We will have to identify a list of independent variables (X) that would be used to predict the outcome of the dependent variable (y). Since this kind of model requires numeric (floats), we will have to do some data transformations first.

Data Transformations

Dependent Variable (y)

We are going to create a new column called **Sold** that would be categorized to show a 0 or 1 value.

In [3]:
```
def sold_val(row):
    if (row['CRNT_LD_STS_TYPE'] == 'Bad') | (row['CRNT_LD_STS_TYPE'] == 'Lost'):
        val = 0.0
    elif (row['CRNT_LD_STS_TYPE'] == 'Sold'):
        val = 1.0

    return val

df_train['SOLD'] =  df_train.apply(sold_val, axis=1)
```

Independent Variables (X)

Our chosen 7 independent variables are – **Make, Body Style, Inventory Type, Lead Media Type, Sale Price, APR and Finance Method**. A lot of trial and error was involved (selecting different combination of features and running the model) that in the interest of time, I am not including in this blog. But be aware this process is very time consuming to select the features that provide the most accurate prediction. Next we need to transform these fields with valid float values. Here are some code examples.

In [10]:
```
def invty_val(row):
    if (row['INVTY_TYPE'] == 'Used'):
        val = 2.0
    elif (row['INVTY_TYPE'] == 'New'):
        val = 1.0
    elif (row['INVTY_TYPE'] == 'CPO'):
        val = 3.0
    else:
        val = 0.0
    return val

df_train['inventory_type'] =  df_train.apply(invty_val, axis=1)
```

Skip to main content

```
In [15]: df_train.CA_VEHICLE_BODY_STYLE.unique()
Out[15]: array(['SUV', 'Sedan', nan, 'Pickup', 'Hatchback', 'Coupe', 'Cargo Van',
                'Passenger', 'Cargo', 'Passenger Van', 'Cutaway', 'Cab & Chassis',
                'Wagon', 'Convertible', 'None', 'Passenger Wagon'], dtype=object)
```

```
In [16]: def type_val(row):
             if (row['CA_VEHICLE_BODY_STYLE'] == 'Sedan') | (row['CA_VEHICLE_BODY_STYLE'] == 'Coupe') | (row['CA_VEHICLE_BODY_STYLE'] == 'Hatchback') | (row['CA_VEHICLE_BO
                 val = 1.0
             elif row['CA_VEHICLE_BODY_STYLE'] == 'SUV' :
                 val = 2.0
             elif row['CA_VEHICLE_BODY_STYLE'] == 'Pickup' :
                 val = 3.0
             elif (row['CA_VEHICLE_BODY_STYLE'] == 'Cargo') | (row['CA_VEHICLE_BODY_STYLE'] == 'Cargo Van')  | (row['CA_VEHICLE_BODY_STYLE'] == 'Cutaway')  | (row['CA_VEHI
                 val = 4.0
             else:
                 val = 5.0
             return val

         df_train['body_style'] =  df_train.apply(type_val, axis=1)
```

```
In [19]: def finance_val(row):
             if (row['FINANCE_TYPE'] == 'Finance'):
                 val = 1.0
             elif row['FINANCE_TYPE'] == 'Lease' :
                 val = 2.0
             elif row['FINANCE_TYPE'] == 'Cash' :
                 val = 3.0
             else:
                 val = 4.0
             return val

         df_train['finance_method'] =  df_train.apply(finance_val, axis=1)
```

Take care of nulls in text columns like the following:

```
In [21]: df_train['SRC_MAKE'] = df_train['SRC_MAKE'].fillna("None")
```

Since prior research has determined that the top 3 selling body styles are SUV, Sedan and Pickup, we will filter out and focus on these body styles only, restricting our dataset to 16 million records (from 23.3 million).

```
In [27]: df_train_stream = df_train[(df_train['body_style'] == 1.0) | (df_train['body_style'] == 2.0) |  (df_train['body_style'] == 3.0)]
```

For the APR and Sale Price columns we will use a scikit-learn library, SimpleImputer method to impute mean values (for the column) for null values (https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html):

```
In [30]: from sklearn.impute import SimpleImputer
         imp_median = SimpleImputer(missing_values=np.nan, strategy='median')
         df_train_stream['apr_adj']= imp_median.fit_transform(df_train_stream[['APR']])
         df_train_stream['sale_price_adj']= imp_median.fit_transform(df_train_stream[['SALE_PRICE']])
```

Building the Model for Predicting Sale of SUV, Sedan and Pickups (July-December, 2019)

We assign dependent (y) and independent variables (X):

```
In [ ]: X = df_train_stream[['SRC_MAKE', 'body_style', 'inventory_type','LD_MEDIA_TYPE','sale_price_adj','apr_adj','finance_method']]
        y = df_train_stream['SOLD']
```

Skip to main content

```
X.sample(5)
```

]:

	SRC_MAKE	body_style	inventory_type	LD_MEDIA_TYPE	sale_price_adj	apr_adj	finance_method
5445186	Hyundai	2.0	1.0	Walk In	28500.0	1.00	4.0
1317703	Dodge	2.0	2.0	Walk In	23993.0	5.99	4.0
21173837	Dodge	3.0	2.0	Internet	23993.0	5.99	1.0
8043202	Jeep	2.0	2.0	Internet	21000.0	14.22	4.0
20943373	Nissan	1.0	1.0	Internet	27860.0	5.99	4.0

We need to convert the Make and Media Type columns to a numeric using the scikit-learn library - Label Encoder (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html?highlight=labelencoder#sklearn.preprocessing.LabelEncoder):

In [36]:
```
from sklearn.preprocessing import LabelEncoder
labelencoder = LabelEncoder()
X.iloc[:, 0] = labelencoder.fit_transform(X.iloc[:, 0])
X.iloc[:, 3] = labelencoder.fit_transform(X.iloc[:, 3])
```

This is the final output of independent variables, now all converted in numeric and float values:

In [37]:
```
X.sample(10)
```

Out[37]:

	SRC_MAKE	body_style	inventory_type	LD_MEDIA_TYPE	sale_price_adj	apr_adj	finance_method
15314694	50	2.0	2.0	1	17588.0	19.14	4.0
18125156	18	1.0	2.0	2	10900.0	5.99	4.0
9745171	113	1.0	2.0	1	6992.0	9.77	4.0
11569446	65	2.0	1.0	1	31220.0	5.99	4.0
21046857	140	2.0	2.0	1	8788.0	5.99	1.0
7682040	53	2.0	1.0	1	46889.0	5.99	2.0
5908684	65	2.0	1.0	5	22820.0	5.99	1.0

Output of the dependent variable also shows binary values only:

Skip to main content

```
In [38]: y.sample(10)
```

```
Out[38]: 10489097    0.0
         13857448    0.0
         2526789     1.0
         11419140    0.0
         17619134    0.0
         2471409     1.0
         10901449    0.0
         668265      1.0
         6435714     0.0
         2338223     1.0
         Name: SOLD, dtype: float64
```

We use scikit-learn library and partition the data into training and test, allowing test size to be 25% and 75% to be used for training purposes.

```
In [39]: from sklearn.model_selection import train_test_split
         from sklearn.linear_model import LogisticRegression
         from sklearn import metrics
         import seaborn as sn
```

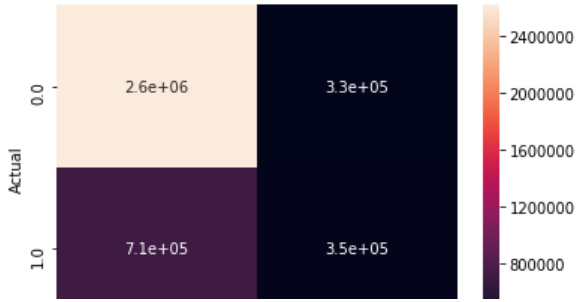
```
In [40]: X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.25,random_state=0)
```

Next we apply Logistic regression and create a Confusion Matrix (https://datatofish.com/confusion-matrix-python/). Predictions are performed on the test set and accuracy is measured.

```
In [ ]: logistic_regression= LogisticRegression()
        logistic_regression.fit(X_train,y_train)
        y_pred=logistic_regression.predict(X_test)
```

```
In [42]: confusion_matrix = pd.crosstab(y_test, y_pred, rownames=['Actual'], colnames=['Predicted'])
         sn.heatmap(confusion_matrix, annot=True)
         print('Accuracy: ', metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy:  0.7390004883542363
```



Skip to main content

The measured accuracy is around 74%, which is not a great result, but if this dataset is supplemented with more accurately captured features, the prediction capability should improve. In any case, it is a good start. 😊 This model can now be used for prediction on newer data sets, for example we can look into Wholesale transactions also.

Hopefully, this is a start in understanding the process involved in creating a ML model. As you can estimate, most of the work is involved up-front in navigating different systems to find the right data and then the ETL process to get it to a better state before we can experiment with models. Please reach out to me if you have questions, comments or need further information. If you interested in this topic, I would encourage you to check out Coursera or EDX courses. Thank you for reading.