Santander Value Prediction Challenge Using BigML

Description

In this competition, Santander Group is asking Kagglers to help them identify the value of transactions for each potential customer. This is a first step that Santander needs to nail in order to personalize their services at scale.

Import Modules

- You will first need to install a number of modules in order to follow along with this notebook.
- Most of these packages, such as numpy and pandas, are available using Anaconda (https://conda.io/docs/user-guide/install/index.html).
- For the machine learning pipeline, we will be making use of the BigML Python bindings (https://bigml.readthedocs.io/en/latest/).

In [8]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
```

In [9]:
```
import sys
print(sys.version)
```

```
3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)]
```

Downloading Data

For this competition, the data is provided in a test and train csv.

In [13]:
```
train = pd.read_csv('data/train.csv')
final_test= pd.read_csv('data/test.csv')
train["target"] = np.log1p(train['target'])
```

In [14]:
```
print(train.shape)
print(final_test.shape)
```

```
(4459, 4993)
(49342, 4992)
```

Data Transformations

1st transformation: Deleting the constant columns; since these columns have an std of 0, they won't have much influence in our model.

Create a list that contains columns to drop.

In [15]:
```
del_col = np.array([])
for series in train.iloc[:,1:].columns:
    if np.std(train[series]) == 0:
       del_col = np.append(del_col, series)
```

Drop the columns that have been decided to be constant in both the test and train.

In [16]:
```
train = train.drop(columns=del_col,axis=1)
final_test = final_test.drop(columns=del_col,axis=1)
```

In [17]:
```
train.head(3)
```

Out[17]:

	id	target	48df886f9	0deb4b6a8	34b15f335	a8cb14b00	2f0771a37	30347e683	d08c
0	000d6aaf2	17.453097	0.0	0	0.0	0	0	0	
1	000fbd867	13.304687	0.0	0	0.0	0	0	0	
2	0027d6b71	16.118096	0.0	0	0.0	0	0	0	

3 rows × 4737 columns

In [18]:
```
final_test.head(3)
```

Out[18]:

	ID	48df886f9	0deb4b6a8	34b15f335	a8cb14b00	2f0771a37	30347e683	d08d1fbe3	6ee
0	000137c73	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
1	00021489f	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0004d7953	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

3 rows × 4736 columns

2nd transformation: Since many columns contain 0s, it will be useful to add a column that sums all values in each row, excluding target and ids, for both test and train. This will help see which customers have a high accumulated score.

In [19]:
```
new_col = train.iloc[:,2:].sum(axis=1)
train.insert(2,"sum_of_values",new_col)

new_col = final_test.iloc[:,1:].sum(axis=1)
final_test.insert(1,"sum_of_values",new_col)
```

In [20]:
```
train.head(3)
```

Out[20]:

	id	target	sum_of_values	48df886f9	0deb4b6a8	34b15f335	a8cb14b00	2f0771a37
0	000d6aaf2	17.453097	7.207683e+08	0.0	0	0.0	0	0
1	000fbd867	13.304687	5.319667e+08	0.0	0	0.0	0	0
2	0027d6b71	16.118096	7.620000e+07	0.0	0	0.0	0	0

3 rows × 4738 columns

In [21]:
```
final_test.head(3)
```

Out[21]:

	ID	sum_of_values	48df886f9	0deb4b6a8	34b15f335	a8cb14b00	2f0771a37	30347e683
0	000137c73	2.188085e+09	0.0	0.0	0.0	0.0	0.0	0.0
1	00021489f	7.480329e+07	0.0	0.0	0.0	0.0	0.0	0.0
2	0004d7953	5.871788e+08	0.0	0.0	0.0	0.0	0.0	0.0

3 rows × 4737 columns

3rd transformation: Adding a new column that counts the number in each row for both test and train. This will give the model an understanding of how many scores a customer has.

In [22]:
```
new_col =  train.iloc[:,3:].apply(lambda x: sum(x.apply(lambda y: 1 if y==0
train.insert(3,"num_of_0s",new_col)

new_col =  final_test.iloc[:,2:].apply(lambda x: sum(x.apply(lambda y: 1 if
final_test.insert(2,"num_of_0s",new_col)
```

In [23]:
```
train.head(3)
```

Out[23]:

	id	target	sum_of_values	num_of_0s	48df886f9	0deb4b6a8	34b15f335	a8cb14b00
0	000d6aaf2	17.453097	7.207683e+08	4633	0.0	0	0.0	0
1	000fbd867	13.304687	5.319667e+08	4668	0.0	0	0.0	0
2	0027d6b71	16.118096	7.620000e+07	4717	0.0	0	0.0	0

3 rows × 4739 columns

In [24]:
```
final_test.head(3)
```

Out[24]:

	ID	sum_of_values	num_of_0s	48df886f9	0deb4b6a8	34b15f335	a8cb14b00	2f0771a37
0	000137c73	2.188085e+09	4663	0.0	0.0	0.0	0.0	0.0
1	00021489f	7.480329e+07	4725	0.0	0.0	0.0	0.0	0.0
2	0004d7953	5.871788e+08	4636	0.0	0.0	0.0	0.0	0.0

3 rows × 4738 columns

4th transformation: Adding new columns that calculate the mean, median, std and max value of each row, excluding target, ids, and newly created columns. Since the rows are made up of multiple separate scores, it will help to add new metrics that explain a row's statistics.

New columns for the train dataset.

In [25]:
```
new_col = train.iloc[:,4:].mean(axis=1)
train.insert(4,"mean",new_col)

new_col = train.iloc[:,5:].median(axis=1)
train.insert(5,"median",new_col)

new_col = train.iloc[:,6:].max(axis=1)
train.insert(6,"max",new_col)

new_col = train.iloc[:,7:].std(axis=1)
train.insert(7,"std",new_col)
```

In [26]:
```
train.head(3)
```

Out[26]:

	id	target	sum_of_values	num_of_0s	mean	median	max	
0	000d6aaf2	17.453097	7.207683e+08	4633	152221.400912	0.0	40000000.0	1.787389e
1	000fbd867	13.304687	5.319667e+08	4668	112347.764874	0.0	50000000.0	1.569893e
2	0027d6b71	16.118096	7.620000e+07	4717	16092.925026	0.0	12000000.0	3.402308e

3 rows × 4743 columns

New columns for the test dataset.

In [27]:
```
new_col = final_test.iloc[:,3:].mean(axis=1)
final_test.insert(3,"mean",new_col)

new_col = final_test.iloc[:,4:].median(axis=1)
final_test.insert(4,"median",new_col)

new_col = final_test.iloc[:,5:].max(axis=1)
final_test.insert(5,"max",new_col)

new_col = final_test.iloc[:,6:].std(axis=1)
final_test.insert(6,"std",new_col)
```

In [28]:
```
final_test.head(3)
```

Out[28]:

std	48df886f9	0deb4b6a8	34b15f335	...	3ecc09859	9281abeea	8675bec0b	3a13ed79a	f677d4d1
3e+07	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.
3e+05	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.
3e+06	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.

Transfer the newly transformed data into CSVs

In [30]:
```
train.to_csv('data/satander_full_train_changed.csv')
final_test.to_csv('data/satander_full_test_changed.csv')
```

Save our BigML Username and Api Key to our environment to access the API

In [31]:
```
import os
os.environ['BIGML_USERNAME'] = "efetoros"
os.environ['BIGML_API_KEY'] = "7e5fc6a649fd0f8517fc8ecf2ebd30151c5d4fb4"
```

Creating our main API object with the input of our project id. The project will enable us tp organize and keep track of our resources created.

In [32]:
```
from bigml.api import BigML
API = BigML(project="project/5b2a78532a83477b000017c1")
```

Importing Data to BigML

In order to start a BigML workflow, a source object has to be created. The API function that creates a source is `create_source` . The method's inputs will be a file path to the csv it will be converting. The source will be created from the csv files written by `to_csv` from before. Many

BigML API functions take in a dictionary with many fields as an additional input. These fields allow for much manipulation of the original function's outcome. We will add the "source_parser" field and set "header" to true.

In [37]:
```
full_source = API.create_source('data/satander_full_train_changed.csv', {"so
final_test_source = API.create_source('data/satander_full_test_changed.csv',
```

BigML's `ok` method is called in order to assure that an object is created and will wait if it is not done being completed.

In [39]:
```
API.ok(full_source)
API.ok(final_test_source)
```

Out[39]:
```
True
```

Creating a Dataset

BigML will use the newly created sources to create datasets which will enable the API to perform many more operations. In order to create a dataset, the API calls the function `create_dataset`. The method will take the sources created by the API as inputs. More details on `create_dataset` can be found at BigML API Dataset Doc. (https://bigml.com/api/datasets#ds_creating_a_dataset).

In [42]:
```
full_dataset = API.create_dataset(full_source)
full_test_dataset = API.create_dataset(final_test_source)
```

In [43]:
```
API.ok(full_dataset)
API.ok(full_test_dataset)
```

Out[43]:
```
True
```

Test-Train Split

Since we want our data to stay in the form of BigML's datasets, the test-train split of the data will be done through BigML's API. This form will allow for the API's computations. The test-train split will be created by the function `create_dataset` mentioned before. However, it will take advantage of the more available inputs of the function. Many BigML API functions take in a dictionary with many fields as an additional input. These fields allow for much manipulation of the original function's outcome. In a test-train split, the field of sample_rate will allow for the choosing of the percentage of data being sampled. The train dataset will have out_of_bag field set to False and the test dataset will have it set to True. Since the test out_of_bag is set to True, its size will be 20% when its sample rate is 80%.

In [44]:
```
train_dataset = API.create_dataset(
    full_dataset, {"name": "Dataset Name | Training",
                   "sample_rate": 0.8, "seed": "my seed"})
train_test_dataset = API.create_dataset(
    full_dataset, {"name": "Dataset Name | Test",
                   "sample_rate": 0.8, "seed": "my seed",
                   "out_of_bag": True})

API.ok(train_dataset)
API.ok(train_test_dataset)
```

Out[44]:
```
True
```

Building an Optimized Ensemble For Full Training Data

BigML's API allows for the creation of many models. For this dataset, an Optimized Ensemble will be used. The BigML API will use the method `create_ensemble` . As stated before, the method takes an additional input of a dictionary. This input allows for the setting of the Objective field, which will be "target" in this model. We will also set "optimize" to True, and let BigML do the heavy lifting.

In [45]:
```
optimal_ensemble = API.create_ensemble(train_dataset,{
    "optimize": True,
    "objective_field":"target"})
```

In [49]:
```
API.ok(optimal_ensemble)
```

Out[49]:
```
True
```

Evaluating our model performance

BigML's API method `create_evaluation` will create an evaluation on BigML models. To conduct an evaluation on the logistic regression model created for this dataset, we will use the model object along with the test dataset as inputs. To extract information from the evaluation, one could use the BigML UI but we will the extract fields from the output evaluation object.

In [50]:
```
evaluation = API.create_evaluation(optimal_ensemble, train_test_dataset)
API.ok(evaluation)
```

Out[50]:
```
True
```

We can confirm by the evaluation object, that our model performs pretty well.

In [51]:
```
evaluation["object"]["result"]["mean"]
```

Out[51]:
```
{'mean_absolute_error': 1.4639, 'mean_squared_error': 3.1522, 'r_square
d': 0}
```

Since we now know the process of creating our model, we will create the same model with the full training data.

Building an Optimized Ensemble for the Full Training Data

In [52]:
```
final_optimal_ensemble = API.create_ensemble(full_dataset,{
    "optimize": True,
    "objective_field":"target"})
```

In [53]:
```
API.ok(final_optimal_ensemble)
```

Out[53]:
```
True
```

Creating a Batch Prediction

The last step of this this workflow will be creating the final batch prediction. This can be done by using BigML's API method `create_batch_prediction` . The method's main inputs will be the model being used and the dataset that the prediction will be based off of. Additional inputs will be in the fields of the dictionary input. For the batch prediction of this dataset, we will set the header and all_fields to True. We will also add the name field for organization.

In [55]:
```
batch_prediction = API.create_batch_prediction(final_optimal_ensemble, full_
    "name": "my batch prediction",
    "all_fields": True,
    "header": True})
API.ok(batch_prediction)
```

Out[55]:
```
True
```

We will use BigML's API method `create_batch_prediction` to download the batch prediction object as a csv.

In [56]:
```
API.download_batch_prediction(batch_prediction,
    filename='data/my_predictions.csv')
```

Out[56]:
```
'data/my_predictions.csv'
```

The last step will be to download the newly created csv into a dataframe, reverse the natural log operation that was done earlier, and finalize the form that Kaggle will need for its final submission.

In [57]:
```
pred = pd.read_csv("data/my_predictions.csv")
```

In [58]:
```
pred['target'] = np.exp(pred['target'])
```

In [59]:
```
pred[["ID","target"]].head()
```

Out[59]:

	ID	target
0	000137c73	4.889008e+06
1	00021489f	1.445015e+06
2	0004d7953	1.936378e+06
3	00056a333	3.485841e+06
4	00056d8eb	8.915707e+05

In [60]:
```
pred[["ID","target"]].to_csv("submission.csv",index=False)
```

All BigML operations are drawn from the BigML API, and full documentation can be found at BigML API Documentation (https://bigml.com/api). This notebook used the API's python bindings, and full documentation can be found at BigML API Python Bindings (https://bigml.readthedocs.io/en/latest/).

In [ ]: