Programming for Big Data
Assignment 2

Course	DT228B
Year	2
Class	Programming for Big Data
Assignment	Spark
Lecturer	Bojan Bozic

Students	**Pierluca Del Buono -** D15128655 **Liam Cantwell -** D16125190 **Ruairí Nugent -** C10335353

Introduction

Kaggle Dataset: https://www.kaggle.com/c/porto-seguro-safe-driver-prediction

Databricks Notebook to create model: https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/1298919323489497/3096462368978933/690406082558930/latest.html

Databricks Notebook to create Kaggle Submission: https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/6292897053021982/1703838843653975/702718371848110/latest.html

Dataset Domain: Car Insurance

Dataset instance: driver policy details for a given year

Binary Target variable: Claim/no Claim

We took our data from Porto Seguro's Safe Driver Prediction competition on Kaggle; Porto Seguro is a major car and home insurance company in Brazil. The aim of their competition was to predict the probability of whether a driver will make an insurance claim, the company then intend to use this data to offer fairer insurance costs based on individual driving habits. Each instance in the dataset relates to the policy details of a driver for a given year.

Dataset Structure.

The dataset had in total 57 features and 1 label to predict. The feature names indicate specific properties:

- "Ind" is referring to individual drivers
- "Reg" is referring to a region
- "Car" refers to the actual car
- "Calc" is a calculated feature.

Below we will look at the summary statistics for individual drivers, region and the car.

The data type was different across the file, while some fields where binary, other represented category, other were ordinal and a small part of the were interval; particular attention was posed in the Feature engineering phase to transform them as binary.

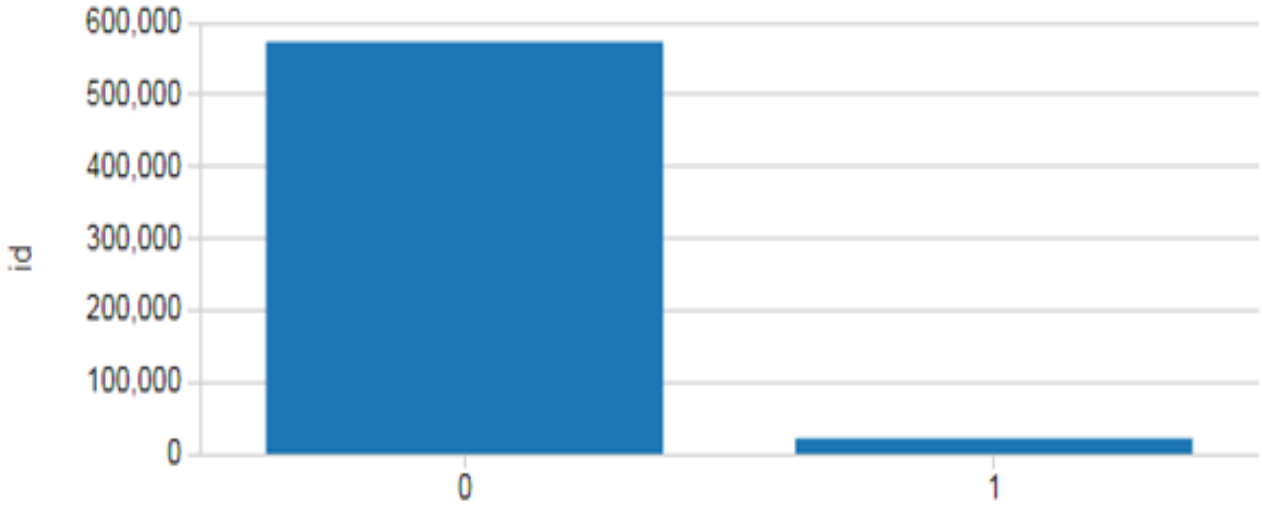
A quick glance at the data shows there is a lot of NAs (missing data), represented by -1 in the fields. a count of missing values reveals there is 846,485 missing values within our training data That equates to approximately 2.5% missing values, and the fact that they are really sparse impacted the predicting algorithm.

Data Statistics

The Dataset has 595,212 rows of data. In the Chapter below is presented an exploratory data analysis, with special focus to the characteristic that will impact the data pre-processing.

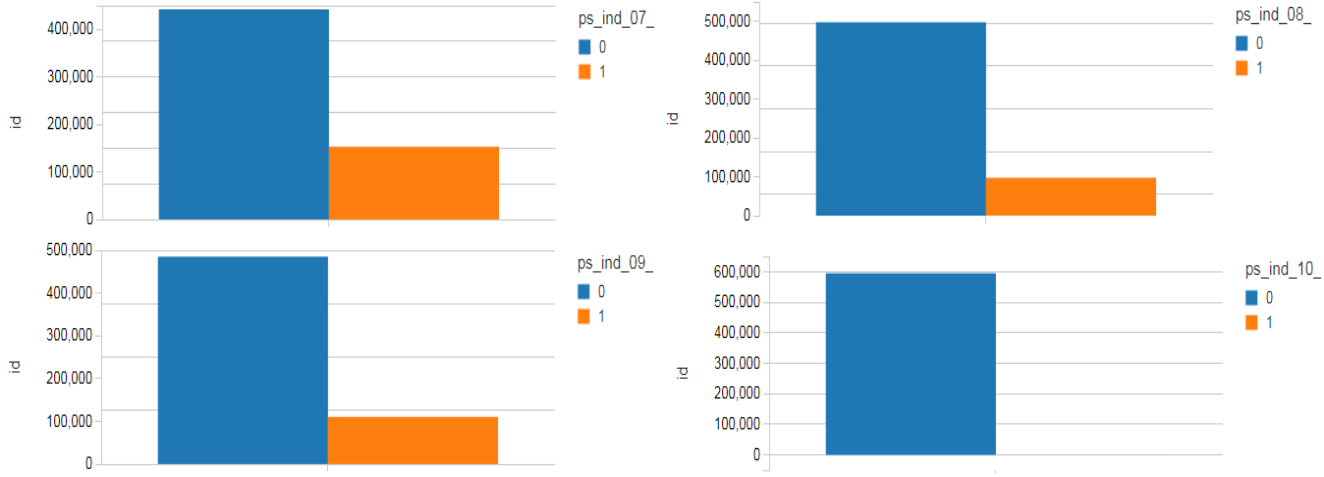
Target variable balance

In the bar chart the distribution of the label or target value. The target value is split 573,518 as 0 (claim not made ~ 96%) and 21,694 as 1 (claim made ~ 4%). This is an important characteristic that lead to the adoption of particular sampling technique.

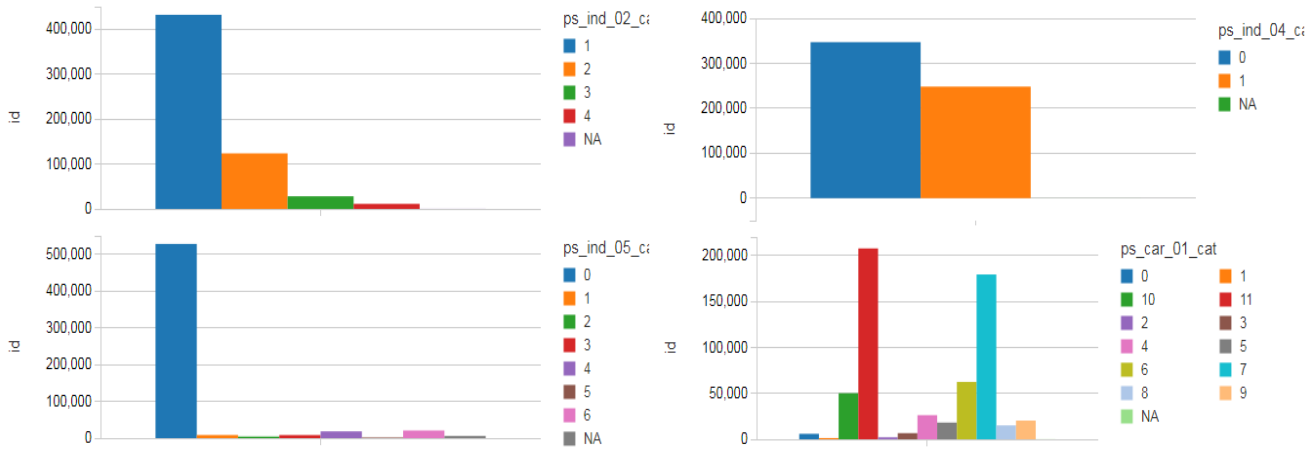


Histogram of features

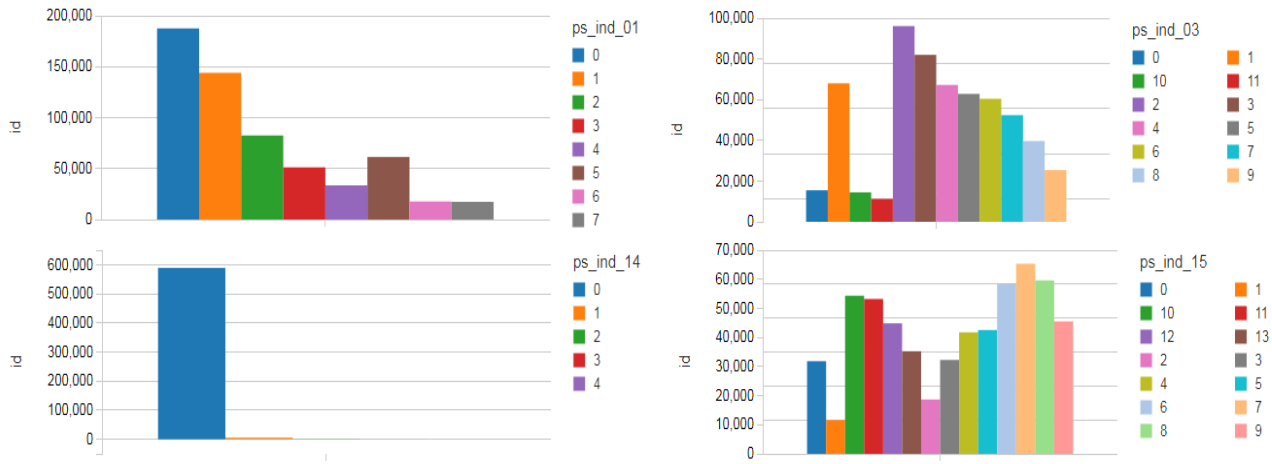
Below Histograms showing the distribution of some of the binary features presented in the dataset; in some case one value is predominant on the other.



Below Histograms related to some representative categorical fields. In some case we have just 2 values, in other multiple values. In the last case one of those might have higher distribution than the other. Strong is the incidence of the not available, represented by -1 in the data set.



In the collection of graphs below we can see similar patterns for the ordinal fields



Missing Data

As mentioned above, we have strong and sparse distribution of missing data, always encode as -1; the bar chart below shows the count of them in the training and in test dataset.

Summary Statistics

The code Snipped below report the total count, mean, SD, range, mode and median for the binary, categorical features.

```
display(train.describe("ps_ind_01","ps_ind_02_cat","ps_ind_03","ps_ind_04_cat","ps_ind_05_cat","ps_ind_05_cat","ps_ind_06_bin","ps_ind_07_bin","ps_ind_08_bin","ps_ind_09_bin","ps_ind_10_bin","ps_ind_11_bin","ps_ind_12_bin","ps_ind_13_bin","ps_ind_14","ps_ind_15","ps_ind_16_bin","ps_ind_17_bin","ps_ind_18_bin"))
```

summary	ps_ind_01	ps_ind_02_cat	ps_ind_03	ps_ind_04_cat	ps_ind_05_cat	ps_ind_05_cat	ps_ind_06_bin	ps_ind_07_bin
count	595212	595212	595212	595212	595212	595212	595212	595212
mean	1.9003783625869775	1.3597990574726553	4.423318078264551	0.4169919462839149	0.41903756852272556	0.41903756852272556	0.39374206165198283	0.25703278831743986
stddev	1.983789117507309	0.6631929091818228	2.69990196069503	0.4930619347689052	1.3500226959148391	1.3500226959148391	0.4885792173105866	0.43699800329779054
min	0	1	0	0	0	0	0	0
max	7	NA	9	NA	NA	NA	1	1

The same is done for regional value, of double format, they will be discretized during pre-processing.

```
display(train.describe("ps_reg_01","ps_reg_02","ps_reg_03"))
```

summary	ps_reg_01	ps_reg_02	ps_reg_03
count	595212	595212	595212
mean	0.6109913778622189	0.43918436784226445	0.8940473269029723
stddev	0.287642619156761	0.4042642867451152	0.34541283872729217
min	0	0	0.061237244
max	0.9	1.8	NA

Below data statistics are calculated for value related to the car.

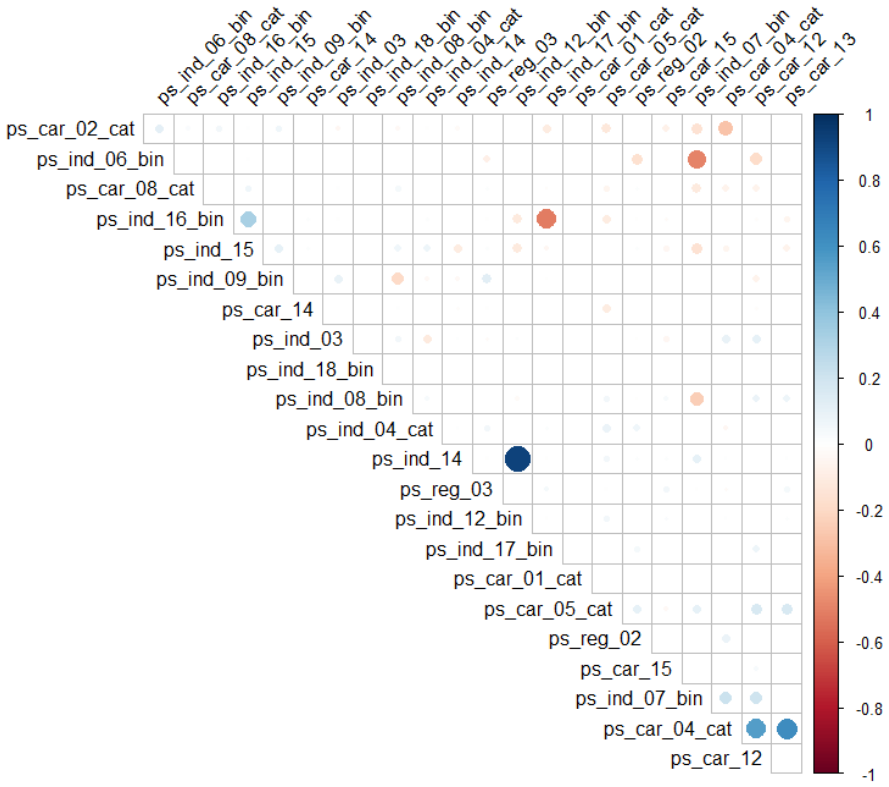
```
display(train.describe("ps_car_01_cat","ps_car_02_cat","ps_car_03_cat","ps_car_04_cat","ps_car_05_cat","ps_car_06_cat","ps_car_07_cat","ps_car_08_cat","ps_car_09_cat","ps_car_10_cat","ps_car_11_cat","ps_car_11","ps_car_12","ps_car_13","ps_car_14","ps_car_15"))
```

summary	ps_car_01_cat	ps_car_02_cat	ps_car_03_cat	ps_car_04_cat	ps_car_05_cat	ps_car_06_cat	ps_car_07_cat	ps_car_08_cat	ps_
count	595212	595212	595212	595212	595212	595212	595212	595212	59
mean	8.297604624394015	0.8299466564081068	0.6017414841749963	0.7251920324187012	0.5253660417907814	6.555339946103238	0.9476207036556723	0.8320799983671293	1.3
stddev	2.505396403072282	0.3756800594049024	0.4895405733725038	2.1534628917823455	0.4993569598612475	5.501444890954262	0.2227909124985248	0.3737954493910166	0.9
min	0	0	0	0	0	0	0	0	0
max	NA	NA	NA	9	NA	9	NA	1	NA

Correlation Matrix

Correlation matrix containing the most highly correlated features was calculated and below is present the R code used to produce it.

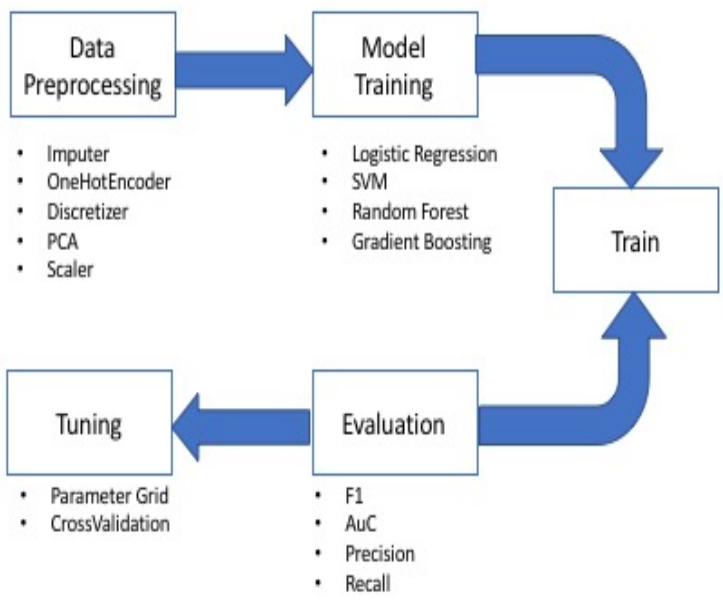
```
include_list<-
c("ps_ind_03","ps_ind_04_cat","ps_ind_06_bin","ps_ind_07_bin","ps_ind_08_bi
n","ps_ind_09_bin","ps_ind_14","ps_ind_15","ps_ind_12_bin","ps_ind_16_bin",
"ps_ind_17_bin","ps_ind_18_bin","ps_reg_02","ps_reg_03","ps_car_01_cat","ps
_car_02_cat","ps_car_04_cat","ps_car_05_cat","ps_car_08_cat","ps_car_12","p
s_car_13","ps_car_14","ps_car_15")

M=cor(train2,use="pairwise.complete.obs")
p = M
p[is.na(M)]=0.2
p[is.na(M)==F]=0
M[is.na(M)]=0
corrplot(M[include_list,include_list],type="upper", method="circle",
is.corr=T, p.mat=p, sig.level=0.01, order = "FPC",diag = FALSE,insig =
"blank",tl.srt = 45,tl.col = "black")
```



There is highly correlation between some values as *ps_ind_12_bin* and *ps_ind_14*, however, by precise choice we decided to include all of them in the pipeline, and have a stage of Feature reduction to limit them.

Machine Learning Algorithm

We needed to learn a supervised classification model so that it can classify whether a person will make a claim or not; all the step that were carried out are represented in the image below.



Data Pre-processing

A series of step where required to clean and to made the dataset suitable for analysis, and in the order where executed:

1. **Imputation**

The dataset contains many values not available sparse across multiple fields, probably representing different category of driver. A logical way to process the data would be to drop the lines with null values(represented by -1). This, however, is not possible, being only 3786 rows without a null value on a total of 595212.
So we proceeded imputing the median value(and not the mean, since many field are categorical) instead of null for all the rows.

```
imput = Imputer(inputCols=seguro.columns
                ,outputCols=seguro.columns)
```

```
imput.setMissingValue(-1)
imput.setStrategy("median")
```

2. OneHotEncoder

Algorithm of classification are working best when categorical fields are each transformed in a series of binary fields. In Spark 2.2 the OneHotEncoder cannot accept multiple fields as input, so the transformation has to be performed separately for any of them. This step must be preceded by a StringIndexer, to transform the format to an index, format accepted as input.
Below a single example of processing.

```
ps_car_01_cat_indexer = StringIndexer()
ps_car_01_cat_indexer.setInputCol("ps_car_01_cat")
ps_car_01_cat_indexer.setOutputCol("ps_car_01_cat_index")
ps_car_01_cat_indexer.setHandleInvalid("skip")
```

```
ps_car_01_cat_encoder = OneHotEncoder()
ps_car_01_cat_encoder.setInputCol("ps_car_01_cat_index")
ps_car_01_cat_encoder.setOutputCol("ps_car_01_cat_feature")
```

3. Discretizer

Algorithm of classification are giving best results when not dealing with continuous fields, so a step discretization to a pre- defined number of buckets has been performed. Then the bucketed fields have gone through the OneHotEncoder.
Below an example of code for just one field.

```
ps_reg_01discretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_reg_01", outputCol="ps_reg_01_disc")
```

```
ps_calc_01encoder = OneHotEncoder()
ps_calc_01encoder.setInputCol("ps_calc_01_disc")
ps_calc_01encoder.setOutputCol("ps_calc_01_disc_vec")
```

4. Scaling

All the features went through a process of centring and standardization of the data, all the data were subtracted from the mean, and divided by their standard deviation in the step below.

```
scaler = StandardScaler()
scaler.setInputCol("features")
scaler.setOutputCol("scaledFeatures")
scaler.setWithStd(True)
scaler.setWithMean(True)
```

Model Estimator

For our model implementation we used a number of different model types. We would then evaluate each type to see which works best for our dataset; the best fit will go through a process of tuning, to improve the results it returns.

A Pipeline with several stage has been defined, how is visible in the appendix, and 4 different model were fitted with the following estimators. The dataset was split in 67% Train portion and 33% Test portion, immediately after its stratified sampling.

1. **Logistic regression**

Logistic Regression of multinomial Type

```
lr = LogisticRegression()
lr.setMaxIter(30)
lr.setFeaturesCol("featurespca")
lr.setLabelCol("target")
lr.setFamily("multinomial")

trainingpipeline.setStages(pstages + [lr])

linear_model = trainingpipeline.fit(train)
```

2. **Random forest**

An ensemble technique based on multiple Decision Tree, executed by the code below.

```
rf = RandomForestClassifier(numTrees=30)
rf.setFeaturesCol("featurespca")
rf.setLabelCol("target")

trainingpipeline.setStages(pstages + [rf])
```

3. **Gradient Boosting**

An ensemble technique based on Decision Tree with minimization of loss function, executed by the code below.

```
gbt = GBTClassifier(maxIter=30)
gbt.setFeaturesCol("featurespca")
gbt.setLabelCol("target")

trainingpipeline.setStages(pstages + [gbt])

gbt_model = trainingpipeline.fit(train)
```

4. **Linear Support Vector Machine**

An algorithm building a hyperplane, minimizing a Hinge loss function with an OWLQN optimizer .

```
svm = LinearSVC(maxIter=30)
svm.setFeaturesCol("featurespca")
svm.setLabelCol("target")

trainingpipeline.setStages(pstages + [svm])

svm_model = trainingpipeline.fit(train)
```

Model Improvement

1. **Stratified Sampling**

The strong unbalance in the feature to be predicted, just 4% on the total, lead to insufficient prediction results in the first phase, so a Stratified Cross Validation with replacement would be appropriate to train the dataset. Spark Ml library doesn't have such step, but allow to sample the dataset.

```
stratified_data = seguro.sampleBy('target', fractions={0: 0.0364, 1:
1.0}).cache()

[train,test] = stratified_data.randomSplit([0.67, 0.33])
```

After stratified sampling the one below was the target distribution in the dataset, that was reduced to 43668 instances.

```
target count
0 1.0    21694
1 0.0    21974
```

2. **Principal Component Analysis**

A PCA step was introduced to remove complexity from the algorithm, and reduce the learning time, keeping the number of features close to 30; we realized similar metrics to the not reduced version, with faster fitting train.

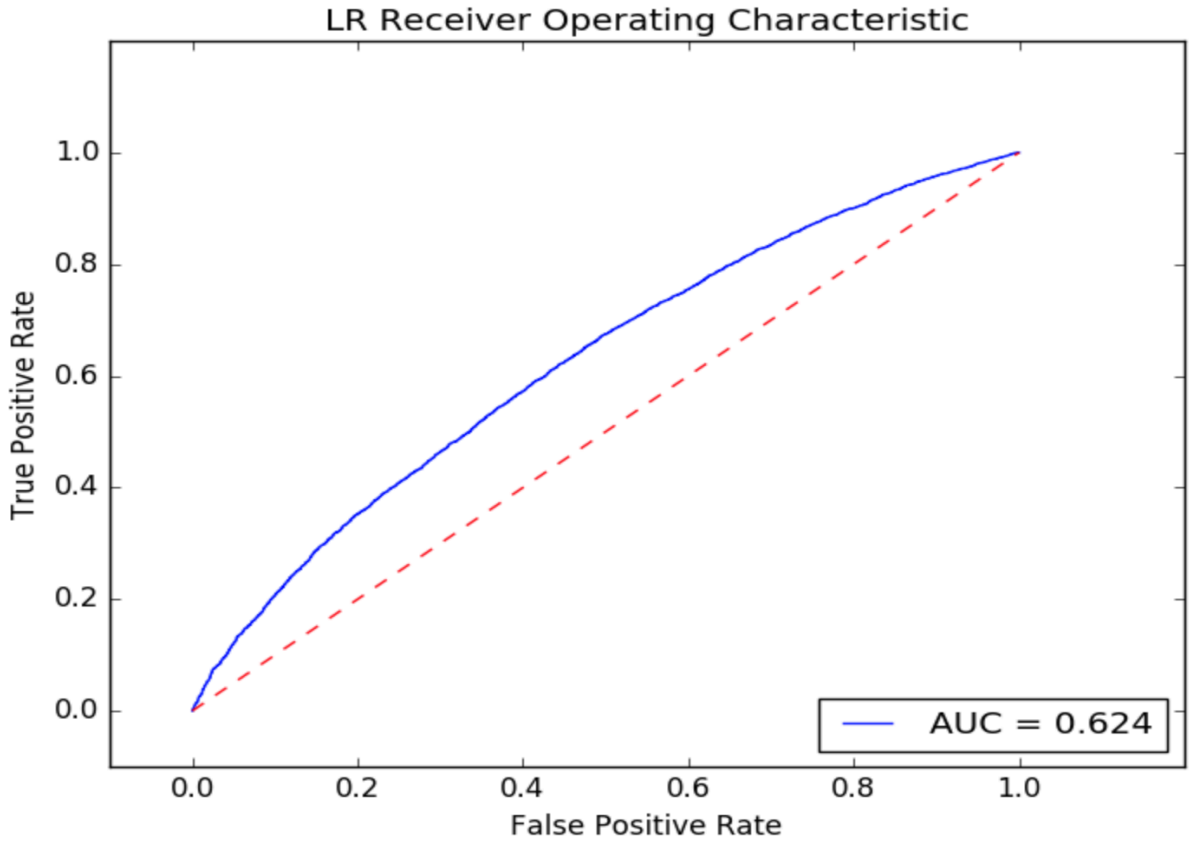
```
pca = PCA()
pca.setInputCol("scaledFeatures")
pca.setOutputCol("featurespca")
pca.setK(30)
```

Evaluation

Evaluation has been performed on the 4 models produced, using the ML library MulticlassClassificationEvaluator and the Mllib library BinaryClassMetrics, that requires dataframes.

	ROC	weighted Recall	weighted Precision	F1	Accuracy
SVM	0.62266	0.586599595	0.586727315	0.58626025	0.5865996
Logistic Regression	0.62435	0.587088661	0.587075937	0.58706208	0.5870887
Random Forest	0.61499	0.584852931	0.584886089	0.58485214	0.5848529
GB Decision Tree	0.60554	0.573324949	0.573327676	0.57318957	0.5733249

Selection is useful done on the ROC parameter, equal the area under the ROC curve, in all the metrics however, there is a single model slightly performing better than the other. Below the ROC curve for that model, a Logistic Regression one. the full code to retrieve them from the 4 models is available in the appendix.



Tuning

The best performing algorithm went through a phase of tuning where a Grid was defined, and all the parameter of the Grid were tested via a Cross-Validation on multiple folds; the optimal number of folds should be 10, but we reduced it to a number of 5, to allow the tuning to complete in a reasonable time.

This was producing the following a model with the following improved metrics:

ROC:	0.62440
weightedRecall:	0.587088660658
weightedPrecision :	0.586727315448
F1:	0.586260247374
Accuracy:	0.586636861089

With this additional metrics, including confusion matrix:

Precision of True	0.587875255028
Precision of False	0.585827405717
Recall of True	0.566573033708
Recall of False	0.606839983317
F-1 Score	0.586809194439
Confusion Matrix	[[4365. 2828.] [3086. 4034.]]

The complete code for metrics extraction is visible in the appendix.

Parameter Grid

A Parameter Grid has been build all the combination to be tried during the tuning process.

```
paramGrid = ParamGridBuilder()\
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.fitIntercept, [False, True])\
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
    .build()
```

Cross Validation

Using ROC value as discriminant, all the combination were weighted, and it was retained on CV model just the one achieving the best value.

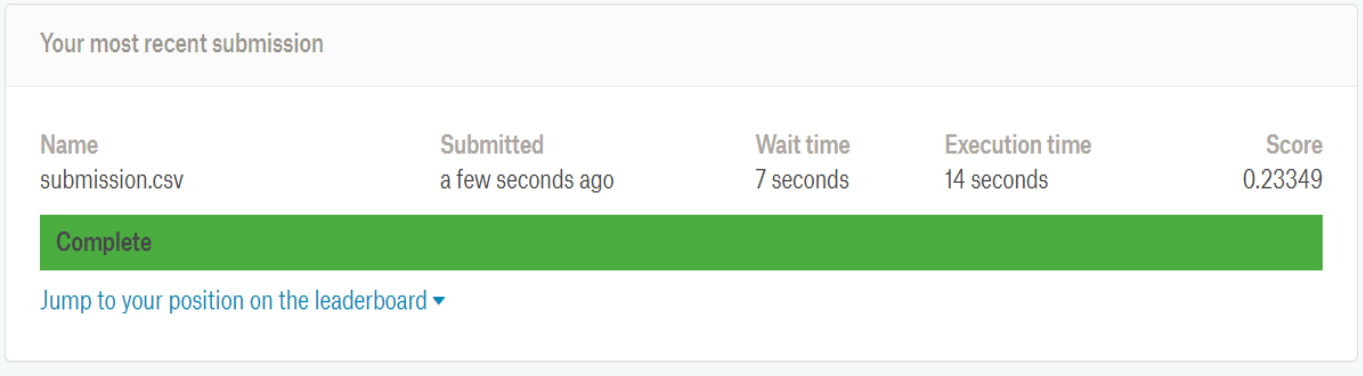
```
evaluator = BinaryClassificationEvaluator()
evaluator.setLabelCol("target")
evaluator.setRawPredictionCol("rawPrediction")
evaluator.setMetricName("areaUnderROC")

pipeline = trainingpipeline.setStages(pstages + [lr])
```

```
cross_val = CrossValidator(estimator=pipeline,
                           estimatorParamMaps=paramGrid,
                           evaluator= evaluator,
                           numFolds=5)

cv_model = cross_val.fit(train)
```

Kaggle Submission

The generated model was used to score the data that does not have any target variable set. A csv file is generated with only the id and prediction values. This was submitted to the Kaggle competition.

	id	target
2	0	0.397
3	1	0.429
4	2	0.482



Discussion

The dataset presented a lot of initial difficulties, being really big, with a high number of features, without a specific description, in different formats and with sparse null values, represented by -1. Moreover, there was a strong unbalance in the label to predict, representing a customer filling the claim.

Sampling stratification was used to achieve results better than the null-classifier, while feature reduction algorithm, Principal Component Analysis, was reducing the complexity and the learning times significantly, keeping the precision lost to a minimum.

A Long section of the code was also focused in Feature engineering, organizing the features in a way that would be easier for the classifier to work with them, favouring the binary encoding.

4 distinct Algorithms were learned, and the one that was performing slightly better was the logistic regression; this algorithm went through a really long tuning phase(the whole tuning

time taking more than 1 hour on a Databricks workbook), but unfortunately the best model achievable didn't improve significantly the results, because of limitation of the cross-fold validation algorithm of Spark, that doesn't allow to use Stratified Sampling picking up its folds.

In summary we setup a pipeline that produced predictions that were significantly better than a baseline, with a ROC of 0.624. Our submission on Kaggle was not very high in the rankings but the types of solutions used to get in the money for this competition were leveraging libraries that allowed very complex algorithms, such as Scikit-learn for Python and Caret for R. It should also be noted that the accuracy of the winning submission was not very high.

Work breakdown

We all had input in all stages but we decided to break up the work as follows:

All: Data selection, approach discussion, report and results discussion.
Ruairi: Investigation and Exploratory Data Analysis.
Pierluca: Model implementation and improvement.
Liam: Evaluation, Tuning and Kaggle competition Submission.

Appendix

```
from pyspark.ml import Pipeline

from pyspark.ml.classification import GBTClassifier
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.classification import LinearSVC

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

from pyspark.ml.feature import Imputer
from pyspark.ml.feature import OneHotEncoder
from pyspark.ml.feature import PCA
from pyspark.ml.feature import QuantileDiscretizer
from pyspark.ml.feature import StandardScaler
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler

from pyspark.sql.types import *

from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.tuning import CrossValidator

# Schema definition
schema=StructType([
    StructField("id", FloatType(), True),
    StructField("target", FloatType(), True),
    StructField("ps_ind_01", FloatType(), True),
    StructField("ps_ind_02_cat", FloatType(), True),
    StructField("ps_ind_03", FloatType(), True),
    StructField("ps_ind_04_cat", FloatType(), True),
    StructField("ps_ind_05_cat", FloatType(), True),
    StructField("ps_ind_06_bin", FloatType(), True),
    StructField("ps_ind_07_bin", FloatType(), True),
    StructField("ps_ind_08_bin", FloatType(), True),
    StructField("ps_ind_09_bin", FloatType(), True),
    StructField("ps_ind_10_bin", FloatType(), True),
    StructField("ps_ind_11_bin", FloatType(), True),
    StructField("ps_ind_12_bin", FloatType(), True),
    StructField("ps_ind_13_bin", FloatType(), True),
    StructField("ps_ind_16_bin", FloatType(), True),
    StructField("ps_ind_17_bin", FloatType(), True),
    StructField("ps_ind_18_bin", FloatType(), True),
    StructField("ps_reg_01", DoubleType(), True),
    StructField("ps_reg_02", DoubleType(), True),
    StructField("ps_reg_03", DoubleType(), True),
    StructField("ps_car_01_cat", FloatType(), True),
```

```
    StructField("ps_car_02_cat", FloatType(), True),
    StructField("ps_car_03_cat", FloatType(), True),
    StructField("ps_car_04_cat", FloatType(), True),
    StructField("ps_car_05_cat", FloatType(), True),
    StructField("ps_car_06_cat", FloatType(), True),
    StructField("ps_car_07_cat", FloatType(), True),
    StructField("ps_car_08_cat", FloatType(), True),
    StructField("ps_car_09_cat", FloatType(), True),
    StructField("ps_car_10_cat", FloatType(), True),
    StructField("ps_car_11_cat", FloatType(), True),
    StructField("ps_car_12", FloatType(), True),
    StructField("ps_car_13", FloatType(), True),
    StructField("ps_car_14", FloatType(), True),
    StructField("ps_car_15", FloatType(), True),
    StructField("ps_calc_01", FloatType(), True),
    StructField("ps_calc_02", FloatType(), True),
    StructField("ps_calc_03", FloatType(), True),
    StructField("ps_calc_04", FloatType(), True),
    StructField("ps_calc_05", FloatType(), True),
    StructField("ps_calc_06", FloatType(), True),
    StructField("ps_calc_07", FloatType(), True),
    StructField("ps_calc_08", FloatType(), True),
    StructField("ps_calc_09", FloatType(), True),
    StructField("ps_calc_10", FloatType(), True),
    StructField("ps_calc_11", FloatType(), True),
    StructField("ps_calc_12", FloatType(), True),
    StructField("ps_calc_13", FloatType(), True),
    StructField("ps_calc_14", FloatType(), True),
    StructField("ps_calc_15_bin", FloatType(), True),
    StructField("ps_calc_16_bin", FloatType(), True),
    StructField("ps_calc_17_bin", FloatType(), True),
    StructField("ps_calc_18_bin", FloatType(), True),
    StructField("ps_calc_19_bin", FloatType(), True),
    StructField("ps_calc_20_bin", FloatType(), True)
])

# file ingestion
seguro = spark.read.csv("/FileStore/tables/train.csv",header=True,
schema=schema)

'''
MissingValueHandler Estimator to replace -1 with the median value.
Mean will introduce impure values.
'''
imput = Imputer(inputCols=seguro.columns
                ,outputCols=seguro.columns)
imput.setMissingValue(-1)
imput.setStrategy("median")
```

```
'''
StringIndexer to encode a category to a indices. On Spark 2.2 we have to
process separately each field
for version after we could use a single method call for all the fields.

'''
ps_car_01_cat_indexer = StringIndexer()
ps_car_01_cat_indexer.setInputCol("ps_car_01_cat")
ps_car_01_cat_indexer.setOutputCol("ps_car_01_cat_index")
ps_car_01_cat_indexer.setHandleInvalid("skip")

ps_car_02_cat_indexer = StringIndexer()
ps_car_02_cat_indexer.setInputCol("ps_car_02_cat")
ps_car_02_cat_indexer.setOutputCol("ps_car_02_cat_index")
ps_car_02_cat_indexer.setHandleInvalid("skip")

ps_car_03_cat_indexer = StringIndexer()
ps_car_03_cat_indexer.setInputCol("ps_car_03_cat")
ps_car_03_cat_indexer.setOutputCol("ps_car_03_cat_index")
ps_car_03_cat_indexer.setHandleInvalid("skip")

ps_car_04_cat_indexer = StringIndexer()
ps_car_04_cat_indexer.setInputCol("ps_car_04_cat")
ps_car_04_cat_indexer.setOutputCol("ps_car_04_cat_index")
ps_car_04_cat_indexer.setHandleInvalid("skip")

ps_car_05_cat_indexer = StringIndexer()
ps_car_05_cat_indexer.setInputCol("ps_car_05_cat")
ps_car_05_cat_indexer.setOutputCol("ps_car_05_cat_index")
ps_car_05_cat_indexer.setHandleInvalid("skip")

ps_car_06_cat_indexer = StringIndexer()
ps_car_06_cat_indexer.setInputCol("ps_car_06_cat")
ps_car_06_cat_indexer.setOutputCol("ps_car_06_cat_index")
ps_car_06_cat_indexer.setHandleInvalid("skip")

ps_car_07_cat_indexer = StringIndexer()
ps_car_07_cat_indexer.setInputCol("ps_car_07_cat")
ps_car_07_cat_indexer.setOutputCol("ps_car_07_cat_index")
ps_car_07_cat_indexer.setHandleInvalid("skip")

ps_car_08_cat_indexer = StringIndexer()
ps_car_08_cat_indexer.setInputCol("ps_car_08_cat")
ps_car_08_cat_indexer.setOutputCol("ps_car_08_cat_index")
ps_car_08_cat_indexer.setHandleInvalid("skip")

ps_car_09_cat_indexer = StringIndexer()
ps_car_09_cat_indexer.setInputCol("ps_car_09_cat")
ps_car_09_cat_indexer.setOutputCol("ps_car_09_cat_index")
```

```
ps_car_09_cat_indexer.setHandleInvalid("skip")

ps_car_10_cat_indexer = StringIndexer()
ps_car_10_cat_indexer.setInputCol("ps_car_10_cat")
ps_car_10_cat_indexer.setOutputCol("ps_car_10_cat_index")
ps_car_10_cat_indexer.setHandleInvalid("skip")

ps_ind_02_cat_indexer = StringIndexer()
ps_ind_02_cat_indexer.setInputCol("ps_ind_02_cat")
ps_ind_02_cat_indexer.setOutputCol("ps_ind_02_cat_index")
ps_ind_02_cat_indexer.setHandleInvalid("skip")

ps_ind_04_cat_indexer = StringIndexer()
ps_ind_04_cat_indexer.setInputCol("ps_ind_04_cat")
ps_ind_04_cat_indexer.setOutputCol("ps_ind_04_cat_index")
ps_ind_04_cat_indexer.setHandleInvalid("skip")

ps_ind_05_cat_indexer = StringIndexer()
ps_ind_05_cat_indexer.setInputCol("ps_ind_05_cat")
ps_ind_05_cat_indexer.setOutputCol("ps_ind_05_cat_index")
ps_ind_05_cat_indexer.setHandleInvalid("skip")

'''
OneHotEncoder transformer to map
indices to a  binary vectors, where each vector.
'''
ps_car_01_cat_encoder = OneHotEncoder()
ps_car_01_cat_encoder.setInputCol("ps_car_01_cat_index")
ps_car_01_cat_encoder.setOutputCol("ps_car_01_cat_feature")

ps_car_02_cat_encoder = OneHotEncoder()
ps_car_02_cat_encoder.setInputCol("ps_car_02_cat_index")
ps_car_02_cat_encoder.setOutputCol("ps_car_02_cat_feature")

ps_car_03_cat_encoder = OneHotEncoder()
ps_car_03_cat_encoder.setInputCol("ps_car_03_cat_index")
ps_car_03_cat_encoder.setOutputCol("ps_car_03_cat_feature")

ps_car_04_cat_encoder = OneHotEncoder()
ps_car_04_cat_encoder.setInputCol("ps_car_04_cat_index")
ps_car_04_cat_encoder.setOutputCol("ps_car_04_cat_feature")

ps_car_05_cat_encoder = OneHotEncoder()
ps_car_05_cat_encoder.setInputCol("ps_car_05_cat_index")
ps_car_05_cat_encoder.setOutputCol("ps_car_05_cat_feature")

ps_car_06_cat_encoder = OneHotEncoder()
ps_car_06_cat_encoder.setInputCol("ps_car_06_cat_index")
ps_car_06_cat_encoder.setOutputCol("ps_car_06_cat_feature")
```

```
ps_car_07_cat_encoder = OneHotEncoder()
ps_car_07_cat_encoder.setInputCol("ps_car_07_cat_index")
ps_car_07_cat_encoder.setOutputCol("ps_car_07_cat_feature")

ps_car_08_cat_encoder = OneHotEncoder()
ps_car_08_cat_encoder.setInputCol("ps_car_08_cat_index")
ps_car_08_cat_encoder.setOutputCol("ps_car_08_cat_feature")

ps_car_09_cat_encoder = OneHotEncoder()
ps_car_09_cat_encoder.setInputCol("ps_car_09_cat_index")
ps_car_09_cat_encoder.setOutputCol("ps_car_09_cat_feature")

ps_car_10_cat_encoder = OneHotEncoder()
ps_car_10_cat_encoder.setInputCol("ps_car_10_cat_index")
ps_car_10_cat_encoder.setOutputCol("ps_car_10_cat_feature")

ps_ind_02_cat_encoder = OneHotEncoder()
ps_ind_02_cat_encoder.setInputCol("ps_ind_02_cat_index")
ps_ind_02_cat_encoder.setOutputCol("ps_ind_02_cat_feature")

ps_ind_04_cat_encoder = OneHotEncoder()
ps_ind_04_cat_encoder.setInputCol("ps_ind_04_cat_index")
ps_ind_04_cat_encoder.setOutputCol("ps_ind_04_cat_feature")

ps_ind_05_cat_encoder = OneHotEncoder()
ps_ind_05_cat_encoder.setInputCol("ps_ind_05_cat_index")
ps_ind_05_cat_encoder.setOutputCol("ps_ind_05_cat_feature")

'''
QuantileDiscretizer Estimator to discretize the continuous values into 10
bins
'''

ps_reg_01discretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_reg_01", outputCol="ps_reg_01_disc")
ps_reg_02discretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_reg_02", outputCol="ps_reg_02_disc")
ps_reg_03discretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_reg_03", outputCol="ps_reg_03_disc")
ps_car_11discretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_car_11_cat", outputCol="ps_car_11_disc")
ps_car_12discretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_car_12", outputCol="ps_car_12_disc")
ps_car_13discretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_car_13", outputCol="ps_car_13_disc")
ps_car_14discretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_car_14", outputCol="ps_car_14_disc")
```

```
ps_car_15discretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_car_15", outputCol="ps_car_15_disc")
ps_calc_01_catdiscretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_calc_01", outputCol="ps_calc_01_disc")
ps_calc_02_catdiscretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_calc_02", outputCol="ps_calc_02_disc")
ps_calc_03_catdiscretizer = QuantileDiscretizer(numBuckets=10,
inputCol="ps_calc_03", outputCol="ps_calc_03_disc")

'''
OneHotEncoder to  one-hot encode vectors from these binned discretized
values.
'''
ps_calc_01encoder = OneHotEncoder()
ps_calc_01encoder.setInputCol("ps_calc_01_disc")
ps_calc_01encoder.setOutputCol("ps_calc_01_disc_vec")

ps_calc_02encoder = OneHotEncoder()
ps_calc_02encoder.setInputCol("ps_calc_02_disc")
ps_calc_02encoder.setOutputCol("ps_calc_02_disc_vec")

ps_calc_03encoder = OneHotEncoder()
ps_calc_03encoder.setInputCol("ps_calc_03_disc")
ps_calc_03encoder.setOutputCol("ps_calc_03_disc_vec")

ps_reg_01encoder = OneHotEncoder()
ps_reg_01encoder.setInputCol("ps_reg_01_disc")
ps_reg_01encoder.setOutputCol("ps_reg_01_disc_vec")

ps_reg_02encoder = OneHotEncoder()
ps_reg_02encoder.setInputCol("ps_reg_02_disc")
ps_reg_02encoder.setOutputCol("ps_reg_02_disc_vec")

ps_reg_03encoder = OneHotEncoder()
ps_reg_03encoder.setInputCol("ps_reg_03_disc")
ps_reg_03encoder.setOutputCol("ps_reg_03_disc_vec")

ps_car_11encoder = OneHotEncoder()
ps_car_11encoder.setInputCol("ps_car_11_disc")
ps_car_11encoder.setOutputCol("ps_car_11_disc_vec")

ps_car_12encoder = OneHotEncoder()
ps_car_12encoder.setInputCol("ps_car_12_disc")
ps_car_12encoder.setOutputCol("ps_car_12_disc_vec")

ps_car_13encoder = OneHotEncoder()
ps_car_13encoder.setInputCol("ps_car_13_disc")
ps_car_13encoder.setOutputCol("ps_car_13_disc_vec")
```

```
ps_car_14encoder = OneHotEncoder()
ps_car_14encoder.setInputCol("ps_car_14_disc")
ps_car_14encoder.setOutputCol("ps_car_14_disc_vec")

ps_car_15encoder = OneHotEncoder()
ps_car_15encoder.setInputCol("ps_car_15_disc")
ps_car_15encoder.setOutputCol("ps_car_15_disc_vec")

'''
scaling and centered all the values with the standard scaler.
'''
scaler = StandardScaler()
scaler.setInputCol("features")
scaler.setOutputCol("scaledFeatures")
scaler.setWithStd(True)
scaler.setWithMean(True)

'''
PCA to reduce complexity and number of features
'''

pca = PCA()
pca.setInputCol("scaledFeatures")
pca.setOutputCol("featurespca")
pca.setK(30)# similar result with 30 that with all, 25 a little less

'''
Below all the steps of the algorithm.
'''

pstages = [
 imput
,ps_car_01_cat_indexer
,ps_car_02_cat_indexer
,ps_car_03_cat_indexer
,ps_car_04_cat_indexer
,ps_car_05_cat_indexer
,ps_car_06_cat_indexer
,ps_car_07_cat_indexer
,ps_car_08_cat_indexer
,ps_car_09_cat_indexer
,ps_car_10_cat_indexer
,ps_ind_02_cat_indexer
,ps_ind_04_cat_indexer
,ps_ind_05_cat_indexer
,ps_car_01_cat_encoder
,ps_car_02_cat_encoder
,ps_car_03_cat_encoder
,ps_car_04_cat_encoder
```

```
,ps_car_05_cat_encoder
,ps_car_06_cat_encoder
,ps_car_07_cat_encoder
,ps_car_08_cat_encoder
,ps_car_09_cat_encoder
,ps_car_10_cat_encoder
,ps_ind_02_cat_encoder
,ps_ind_04_cat_encoder
,ps_ind_05_cat_encoder
,ps_reg_01discretizer
,ps_reg_02discretizer
,ps_reg_03discretizer
,ps_car_11discretizer
,ps_car_12discretizer
,ps_car_13discretizer
,ps_car_14discretizer
,ps_car_15discretizer
,ps_calc_01_catdiscretizer
,ps_calc_02_catdiscretizer
,ps_calc_03_catdiscretizer
,ps_calc_01encoder
,ps_calc_02encoder
,ps_calc_03encoder
,ps_reg_01encoder
,ps_reg_02encoder
,ps_reg_03encoder
,ps_car_11encoder
,ps_car_12encoder
,ps_car_13encoder
,ps_car_14encoder
,ps_car_15encoder
,assembler
,scaler
,pca]

trainingpipeline = Pipeline()

#stratification test ratio is 1: 0.0364 and 0: 0.964
stratified_data = seguro.sampleBy('target', fractions={0: 0.0364, 1:
1.0}).cache()

#Train Test Splitting
[train,test] = stratified_data.randomSplit([0.67, 0.33])

'''
Fitting the 4 types of Algorithm
'''

lr = LogisticRegression()
```

```
lr.setMaxIter(30)
lr.setFeaturesCol("featurespca")
lr.setLabelCol("target")
lr.setFamily("binomial")

trainingpipeline.setStages(pstages + [lr])

linear_model = trainingpipeline.fit(train)

rf = RandomForestClassifier(numTrees=30)
rf.setFeaturesCol("featurespca")
rf.setLabelCol("target")

trainingpipeline.setStages(pstages + [rf])

rf_model = trainingpipeline.fit(train)

gbt = GBTClassifier(maxIter=30)
gbt.setFeaturesCol("featurespca")
gbt.setLabelCol("target")

trainingpipeline.setStages(pstages + [gbt])

gbt_model = trainingpipeline.fit(train)

svm = LinearSVC(maxIter=30)
svm.setFeaturesCol("featurespca")
svm.setLabelCol("target")

trainingpipeline.setStages(pstages + [svm])

svm_model = trainingpipeline.fit(train)

linear_testDataPredictions = linear_model.transform(test)
rf_testDataPredictions = rf_model.transform(test)
gbt_testDataPredictions = gbt_model.transform(test)

evaluator = MulticlassClassificationEvaluator()
'''
utilize the MulticlassClassificationEvaluator for computing
the evaluation.
'''
evaluator = MulticlassClassificationEvaluator()
evaluator.setLabelCol("target")
evaluator.setPredictionCol("prediction")
evaluator.setMetricName("accuracy")

accuracy = evaluator.evaluate(linear_testDataPredictions)
print("Accuracy lr: {0}".format(accuracy))
```

```
accuracy = evaluator.evaluate(rf_testDataPredictions)
print("Accuracy rf: {0}".format(accuracy))

accuracy = evaluator.evaluate(gbt_testDataPredictions)
print("Accuracy gbt: {0}".format(accuracy))

accuracy = evaluator.evaluate(svm_testDataPredictions)
print("Accuracy svm: {0}".format(accuracy))

evaluator = MulticlassClassificationEvaluator()
evaluator.setLabelCol("target")
evaluator.setPredictionCol("prediction")
evaluator.setMetricName("f1")

f1 = evaluator.evaluate(linear_testDataPredictions)
print("f1 lr: {0}".format(f1))

f1 = evaluator.evaluate(rf_testDataPredictions)
print("f1 rf: {0}".format(f1))

f1 = evaluator.evaluate(gbt_testDataPredictions)
print("f1 gbt: {0}".format(f1))

f1 = evaluator.evaluate(svm_testDataPredictions)
print("f1 svm: {0}".format(f1))

evaluator = MulticlassClassificationEvaluator()
evaluator.setLabelCol("target")
evaluator.setPredictionCol("prediction")
evaluator.setMetricName("weightedPrecision")

weightedPrecision = evaluator.evaluate(linear_testDataPredictions)
print("weightedPrecision lr: {0}".format(weightedPrecision))

weightedPrecision = evaluator.evaluate(rf_testDataPredictions)
print("weightedPrecision rf: {0}".format(weightedPrecision))

weightedPrecision = evaluator.evaluate(gbt_testDataPredictions)
print("weightedPrecision gbt: {0}".format(weightedPrecision))

weightedPrecision = evaluator.evaluate(svm_testDataPredictions)
print("weightedPrecision svm: {0}".format(weightedPrecision))

evaluator = MulticlassClassificationEvaluator()
evaluator.setLabelCol("target")
evaluator.setPredictionCol("prediction")
evaluator.setMetricName("weightedRecall")
```

```
weightedRecall = evaluator.evaluate(linear_testDataPredictions)
print("weightedRecall lr: {0}".format(weightedRecall))

weightedRecall = evaluator.evaluate(rf_testDataPredictions)
print("weightedRecall rf: {0}".format(weightedRecall))

weightedRecall = evaluator.evaluate(gbt_testDataPredictions)
print("weightedRecall gbt: {0}".format(weightedRecall))

weightedRecall = evaluator.evaluate(svm_testDataPredictions)
print("weightedRecall svm: {0}".format(weightedRecall))

'''
We utilize the BinaryClassificationEvaluator for computing
the evaluation. This evaluator by default computes the
'areaUnderROC'
'''
evaluator = BinaryClassificationEvaluator()
evaluator.setLabelCol("target")
evaluator.setRawPredictionCol("rawPrediction")
evaluator.setMetricName("areaUnderROC")

lr_roc_test = evaluator.evaluate(linear_testDataPredictions)
print("Test Linear ROC: {0:.5f}".format(lr_roc_test))

rf_roc_test = evaluator.evaluate(rf_testDataPredictions)
print("Test Random Forest ROC: {0:.5f}".format(rf_roc_test))

gbt_roc_test = evaluator.evaluate(gbt_testDataPredictions)
print("Test GBT ROC: {0:.5f}".format(gbt_roc_test))

svm_roc_test = evaluator.evaluate(svm_testDataPredictions)
print("Test SVM ROC: {0:.5f}".format(svm_roc_test))

'''
Drawing the 4 ROC curves for the distinc results.
'''

predresscore =
gbt_testDataPredictions.withColumn("score",scorepickfuncudf(gbt_testDataPre
dictions["probability"]))

false_positive_rate, true_positive_rate, thresholds = roc_curve(labels,
probs)
roc_auc = auc(false_positive_rate, true_positive_rate)

fig, ax = plt.subplots()
plt.title('GBT Receiver Operating Characteristic')
```

```
plt.plot(false_positive_rate, true_positive_rate, 'b', label='AUC = %0.3f'%
roc_auc)
plt.legend(loc='lower right')
plt.plot([0,1],[0,1],'r--')
plt.xlim([-0.1,1.2])
plt.ylim([-0.1,1.2])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')

display(fig)

'''
Tuning Phase, to perform it on the best algorithm.
'''

paramGrid = ParamGridBuilder()\
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.fitIntercept, [False, True])\
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
    .build()

evaluator = BinaryClassificationEvaluator()
evaluator.setLabelCol("target")
evaluator.setRawPredictionCol("rawPrediction")
evaluator.setMetricName("areaUnderROC")

pipeline = trainingpipeline.setStages(pstages + [lr])

cross_val = CrossValidator(estimator=pipeline,
                           estimatorParamMaps=paramGrid,
                           evaluator= evaluator,
                           numFolds=5)

cv_model = cross_val.fit(train)

bestModel = cv_model.bestModel

bestModel.save("lrSeguro20180330")

#bestModel = LogisticRegression.load("lrSeguro20180330")

best_testDataPredictions = bestModel.transform(test)

evaluator = BinaryClassificationEvaluator()
evaluator.setLabelCol("target")
evaluator.setRawPredictionCol("rawPrediction")
evaluator.setMetricName("areaUnderROC")

lr_roc_test = evaluator.evaluate(best_testDataPredictions)
```

```
print("Test Linear ROC: {0:.5f}".format(lr_roc_test))

evaluator = MulticlassClassificationEvaluator()
evaluator.setLabelCol("target")
evaluator.setPredictionCol("prediction")
evaluator.setMetricName("weightedRecall")

weightedRecall = evaluator.evaluate(linear_testDataPredictions)
print("weightedRecall lr: {0}".format(weightedRecall))

weightedRecall = evaluator.evaluate(linear_testDataPredictions)
print("weightedRecall lr: {0}".format(weightedPrecision))

evaluator.setMetricName("weightedPrecision")

weightedRecall = evaluator.evaluate(linear_testDataPredictions)
print("weightedRecall lr: {0}".format(f1))

evaluator.setMetricName("f1")

accuracy = evaluator.evaluate(best_testDataPredictions)
print("Accuracy lr: {0}".format(accuracy))
```