Microsoft Malware Prediction Challenge in the Cloud

Final Report

Sergio E. Betancourt (sergio.betancourt@mail.utoronto.ca)

2019-06-30

Contents

1 Introduction

Cybersecurity remains a priority for individuals and organizations since the World Wide Web (WWW) was launched to the public in the early 90s. Cyber threats are reported to continue to grow at a fast pace while firms continue to invest in preventive (instead of purely reactive) measures. The following are some worrisome figures to illustrate the economic and security impact of cybersecurity threats:

- 12 billion records/docs stolen globally in 2018 (Juniper Research)
- 60 million Americans affected by identity theft in 2018 (The Harris Poll)
- YoY IoT-related attacks doubled in 2018 (26% of breaches caused by an unsecured IoT device or IoT application).

In this project I examine the Kaggle **2018 Microsoft Malware Challenge** employing a joint Cloud and machine learning toolkit.

2 Methods

The task a hand is outlined in the official competition **website**. It is a binary classification problem over millions of observations, each pertaining to a distinct Windows device. By classifying correctly which device has the highest chance of acquiring malware in the coming time period, we can get an idea of the most influential factors towards said infection.

The dataset contains telemetric information, all of which is described in this report's **Appendix: Dataset Variables and Definitions.**

2.1 Primary Question

Our primary research question is: *Given a set of telemetric features on Windows machines, can we create an effective ML classifier model in the Cloud?*

2.2 Cloud Specifications and Set-up

I took on the extra challenge of setting up this project in a Cloud platform apart from the faculty-provisioned **Queen's** cluster.

The main providers out there are: **Amazon Web Services** (AWS), **Google Cloud Platform** (GCP), and **Microsoft Azure**. I settled for the last two due to the fact that they include one-click Spark cluster set-up services. This proved to be extremely helpful as configuring a cluster from scratch is an extremely difficult and time-consuming task.

Here is a small guide on GCP and Azure:

	GCP	**Azure**
Free Credits	$300	$260
Hadoop/Spark Service	Dataproc	HDInsight
Storage Service	GCloud	Blob/Data Lake
Advantages	SW customization	User-friendly and transparent billing

Ultimately we harnessed the three environments below:

	Queens	**GCP**	**Azure**
HW	4 nodes / 88 cores	3 nodes / 6 cores	6 nodes / 40 cores
Total Memory (GB)	736	45	336
Comments	Great parallelization but mem limit	Least useful	Autoscaling

2.3 Data Collection and Preparation

The following table provides an overview of the final dataset:

MachineIdentifier	EngineVersion	Processor	ChassisTypeName	HasDetections
b7d94d8f4ccb319768e93e6a35408f65	1.1.15100.1	x86	desktop	1
c88a1f054653ef53617f7fa91f153dc2	1.1.15100.1	x64	desktop	1
24285cb0a4c83b8d84895eed105e8ed3	1.1.15100.1	x64	portable	1
0ccc2aae1969b32322ca6658d959525f	1.1.15300.5	x64	unknown	0
3e9461ee066bb214b6aebd8732efee03	1.1.15000.2	x86	other	0

It is important to note that this dataset is balanced in the dependent variable **HasDetections**. This variable represents the ground truth, which allows us to consider this as a supervised learning problem.

The size of the original dataset is about five gigabytes, containing approximately nine million records and 82 raw features. Most of these features are categorical with a large number of distinct labels. A substantial portion of features also contain plenty of missing values.

The data cleanup and preparation was performed on the original large dataset for consistency. First I calculated the number of missing values per feature, as well as the number of distinct categories for every categorical feature. I discarded those variables with 40% missing values (e.g., PuaMode), those with an excessive number of distinct labels, and those that displayed excessive imbalance. Then I proceeded to group and aggregate the scarce labels in the remaining categorical variables into larger categories to improve numerical stability.

As it pertains to missing values, I created a new category when appropiate, which allowed for stable encoding and processing. There are many procedures to handle missing values (KNN, EM, etc), but for speed's sake I judged the imputation from rudimentary inspection.

One of the challenges of variable selection in classification is the limited number of systematic tools, unlike with regression when one can inspect correlation plots and carry out selection algorithms. I did not worry about variable selection here as there were only 61 features used in our modeling, compared to the hundreds of thousands data points used for training. The greatest concern in this challenge is feature information and numerical stability.

3 Modeling

We consider the below four models for our balanced classification task:

- **Logistic Regression**
- **Support Vector Machine**
- **Random Forests**
- **Gradient-Boosted Trees**

3.1 Logistic Regression Redux

For the sake of my educational background and peace of mind I shall describe in detail the logistic regression model only. For all other models please refer to the very excellent (Hastie, Tibshirani, and Friedman 2009).

The logistic model suits binary outcome variables and my goal in this project is to estimate the probability of a computer having or lacking malware detections as a linear combination of the predictor variables.

Denoting the probability of observing a computer with malware given X features as $\pi = \text{P(Y=1|X)}$ we have:

$$\text{logit}(\pi) = \text{X}\beta \iff \pi = \frac{1}{1 + \exp\{-\text{X}\beta\}} \tag{1}$$

For the problem at hand I consider p+1 parameters in the model—one for each feature in the dataset, plus an intercept. Given the large amount of data available, as well as the focus on maximizing AUROC $\in [0, 1]$ among all candidate models, I apply elastic-net regularization to constrain the magnitude of my model parameters and improve generalization.

For my choice of loss function $\mathcal{L}$, elastic-net plays the following role in model training as we solve the loss minimization problem:

$$\underset{\beta}{\text{argmin}} \{\mathcal{L} + \lambda\mathcal{E}_\alpha\} \text{ s.t. } \mathcal{E}_\alpha \leq t \text{ and } \lambda \in [0, \infty) \tag{2}$$

$$\text{where } \mathcal{E}_\alpha = \left(\alpha \sum_{j=1}^{p+1} |\hat{\beta}_j| + (1-\alpha) \sum_{j=1}^{p+1} \hat{\beta}_j^2 \right), \ \alpha \in [0, 1] \tag{3}$$

3.2 Model Tuning

To improve the performance of my models with respect to test-set AUROC (Bradley 1997) (closing the gap between train-test performance while also obtaining the lowest possible test metric) I perform 3-fold cross validation. K-fold validation is a very effective hyperparameter searching technique, and I would like to acknowledge the limited the number of folds in this project due to limited time and hardware resources.

3.3 Implementation Strategy

New to the Spark framework and both Scala and pySpark APIs, I decided to adopt the following implementation strategy for my modeling effort:

1. Get one model working from first principles using the `pyspark.ml` library, on a small subset of the data, without hyperparameter control
2. Once working, implement pipelining with cross-validation on a bigger portion of the dataset
3. Extract metrics and hyperparameters of interest
4. Build other models with template developed from the above 2. and 3.

It is also important to compare all trained/considered models in terms of AUROC **and** runtime, for many production applications have a time and resource requirement.

4 Results

I measured the success of my models by their train/test Area under the Receiver Operating Curve (AUROC). This is the standard metric for balanced, classification tasks, for which a value of 1 corresponds to perfect classification, 0.5 corresponds to random guessing, and 0 corresponds to the most imperfect results. A high difference in test and train AUROCs (training variance) means that the model may be underfitting and perhaps requires better hyperparameter values, whereas values that are too close require a more thought to identify possible issues.

For all of these models I employed 3-fold cross validation and a 90-10 train-test split. Moreover, they were trained with 3 Spark executors, 95GB memory each, for a total of 225GB of computation memory. Model parallelization was set at level 8 across the board.

Model	Data.size	Optimal.hypers	Train.Test.AUROC	Runtime
SVM	100K (100MB)	fitIntercept: True, regParam/L2: 0.025	[0.65, 0.62]	~4 hrs
LR	500K (500MB)	regParam: 0.025, ElasticParam: 0.0	[0.68, 0.67]	~3 hrs
RF	500K (500MB)	numTrees: 100, maxDepth: 12	[0.66, 0.65]	~6 hrs
GBT	**500K (500MB)**	**maxDepth: 8,maxBins: 25, stepSize: 0.1**	**[0.69, 0.68]**	**~10 hrs**

In the above we see that the top performer in the metric of interest is the GBT (`pyspark.ml.classification.GBTClassifier`); nonetheless it also has the longest traintime. The second best performer is the logistic regression model (`pyspark.ml.classification.LogisticRegression`). This model does have the fastest traintime.

For classification tasks with conventional tabular data structures it seems that boosted machines and ensemble methods have become the norm (in Kaggle competitions and in industrial applications). In our results, the small performance differential of one percentage point in AUROC between the winner and loser contrasts the dramatic difference in their train time. I believe that the LR performed quite well here due to the relatively low number of parameters (61 + intercept.

5 Discussion

Overall this project was incredibly iluminating. I got what I wanted from it:

- Exposure to both domestic (Queen's) and third-party (GCP and Azure) distributed storage and computing systems
- Pipelined implementation of four machine learning models in (py)Spark
- Practice handling a dataset in the gigabytes scale

Nonetheless, even though the instructor and I were quite impressed with the performance of the logistic regression and the improvement achieved with the Gradient-Boosted trees model, the Kaggle challenge winners achieved an AUROC score of .71, with second and third place also achieving .71 at lower decimal values. What this illustrates is that developing ultra high-performance models requires plenty of creativity and hard work, and there are a large number of techniques that may result in marginally (or hopefully dramatically) better results. Something to acknowledge in these competitions, too, is the limited scope for assessment–I believe that these models should be assessed holisticaly, as something that takes unreasonable hours and computation to train, while only offering a "marginal" improvement in one single metric, may not be what a client really can implement in real life.

A natural step after implementing classification models successfully is to consider the case of data imbalanced in the label/predicted variable. This situation requires new techniques, either data (sampling) or model-based, to prevent assigning too much weight to the overrepresented class during training. Moreover, restricting ourselves to a classification framework, the criteria we use to judge model performance must change, too.

In the case of classification in an imbalanced setting, AUROC is often optimistic and does not reflect accurately our model performance, often yielding too optimistic a picture. The natural alternative is the Area under the Precision-Recall Curve (AUPRC).

Recycling one of our best performing models, the logistic regression can be readily adjusted to address the imbalanced case with the addition of the following:

- **Rebalancing scheme**: Let R be the re-balancing ratio in a given two-class dataset, N be the total number of observations, and X be the size of the overrepresented class. Then,

  $$R = \frac{X}{N}$$

  is the implementation of this principle as a new column to be added to the dataset, where every observation from the underrepresented class shall get a score of R, while the others get a score of $1 - R$. Such a new column can be readily fed to `pyspark.ml.Classification.LogisticRegression` as an extra input `weightCol`.
- **Stratified Cross-validation**: the default implementation of K-fold cross-validation in pySpark samples subsets from the training set randomly. However, in order to control for imbalance, we should preserve the imbalance ratio found in the original training set through all folds of the cross-validation step. You can see my implementation of this and the above in **Appendix: Imbalanced Data (Extra)**.

Lastly I would like to mention a few things worth trying in similar future projects with similar datasets and research objectives:

- Bayesian hyperparameter optimization (Snoek, Larochelle, and Adams 2012)
- Bayesian inference with Monte Carlo estimation
- Model or loss-based imbalance techniques beyond under/oversampling and reweighting: Focal cross-entropy loss, etc.
- Faster implementations of GBMs: XGBoost (T. Chen and Guestrin 2016) and lightGBM (Ke et al. 2017)
- Deep learning implementation: MLP in pyspark.ml, etc.

6 References

Bradley, Andrew P. 1997. "The Use of the Area Under the Roc Curve in the Evaluation of Machine Learning Algorithms." *Pattern Recogn.* 30 (7). New York, NY, USA: Elsevier Science Inc.: 1145–59. doi:10.1016/S0031-3203(96)00142-2.

Chen, Tianqi, and Carlos Guestrin. 2016. "XGBoost: A Scalable Tree Boosting System." *CoRR* abs/1603.02754. http://arxiv.org/abs/1603.02754.

Hastie, T., R. Tibshirani, and J. Friedman. 2009. *The Elements of Statistical Learning*. Springer, New York.

Ke, Guolin, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." In *NIPS*.

Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams. 2012. "Practical Bayesian Optimization of Machine Learning Algorithms." In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, 2951–9. NIPS'12. USA: Curran Associates Inc. http://dl.acm.org/citation.cfm?id=2999325.2999464.

7 Appendix: Dataset Variables and Definitions

Feature	Description	Type
MachineIdentifier	Individual machine ID	String
ProductName	Defender state information e.g. win8defender	Categorical
EngineVersion	Defender state information e.g. 1.1.12603.0	Categorical
AppVersion	Defender state information e.g. 4.9.10586.0	Categorical
AvSigVersion	Defender state information e.g. 1.217.1014.0	Categorical
IsBeta	Defender state information e.g. false	Binary
RtpStateBitfield	NA	Numeric
IsSxsPassiveMode	NA	Binary
DefaultBrowsersIdentifier	ID for the machine's default browser	Categorical
AVProductStatesIdentifier	ID for the specific configuration of a user's antivirus software	Categorical
AVProductsInstalled	NA	Numeric
AVProductsEnabled	NA	Numeric
HasTpm	True if machine has tpm	Binary
CountryIdentifier	ID for the country the machine is located in	Categorical
CityIdentifier	ID for the city the machine is located in	Categorical
OrganizationIdentifier	ID for the organization the machine belongs in	Categorical
GeoNameIdentifier	ID for the geographic region a machine is located in	Categorical
LocaleEnglishNameIdentifier	English name of Locale ID of the current user	Categorical
Platform	Calculates platform name	Categorical
Processor	This is the process architecture of the installed operating system	Categorical
OsVer	Version of the current operating system	Categorical
OsBuild	Build of the current operating system	Categorical
OsSuite	Product suite mask for the current operating system.	Categorical
OsPlatformSubRelease	Returns the OS Platform sub	Categorical
OsBuildLab	Build lab that generated the current OS	Categorical
SkuEdition	Uses the Product Type defined in the MSDN to map to a 'SKU	Categorical
IsProtected	Calculated field derived from the Spynet Report's AV Products.	Binary
AutoSampleOptIn	SubmitSamplesConsent value passed in from the service	Binary
PuaMode	Pua Enabled mode from the service	Binary
SMode	This field is set to true when the device is known to be in 'S Mode'	Binary
IeVerIdentifier	NA	Categorical
SmartScreen	This is the SmartScreen enabled string value from registry.	Categorical
Firewall	true (1) for Windows 8.1 and above if firewall is enabled.	Binary
UacLuaenable	Whether "administrator in Admin Approval Mode" is enabled.	Binary
MDC2FormFactor	A grouping based on Device Census level HW characteristics.	Categorical
DeviceFamily	AKA DeviceClass.	Categorical
OEMNameIdentifier	NA	Categorical
OEMModelIdentifier	NA	Categorical
ProcessorCoreCount	Number of logical cores in the processor	Numeric
ProcessorManufacturerId	NA	Categorical
ProcessorModelIdentifier	NA	Categorical
ProcessorClass	A classification of processors into high/medium/low.	Categorical
PrimaryDiskTotalCapacity	Amount of disk space on primary disk of the machine.	Numeric
PrimaryDiskTypeName	Friendly name of Primary Disk Type	Numeric
SystemVolumeTotalCapacity	The size of the partition that the System volume is installed on.	Numeric
HasOpticalDiskDrive	True indicates that the machine has an optical disk drive (CD/DVD)	Binary
TotalPhysicalRAM	Retrieves the physical RAM in MB	Numeric
ChassisTypeName	Numeric representation of what type of chassis the machine has.	Categorical
InternallDisplaySizeInInches	Physical diagonal length in inches of the primary display	Numeric
InternalResolutionHorizontal	Number of pixels in the horizontal direction of the internal display.	Numeric
InternalResolutionVertical	Number of pixels in the vertical direction of the internal display	Numeric

Feature	Description	Type
PowerPlatformRoleName	Indicates the OEM preferred power management profile.	Categorical
InternalBatteryType	NA	Categorical
InternalBatteryNumberCharges	NA	Numeric
OSVersion	Numeric OS version Example	Categorical
OSArchitecture	Architecture on which the OS is based.	Categorical
OSBranch	Branch of the OS extracted from the OsVersionFull.	Categorical
OSBuildNumber	OS Build number extracted from the OsVersionFull.	Categorical
OSBuildRevision	OS Build revision extracted from the OsVersionFull.	Categorical
OSEdition	Edition of the current OS.	Categorical
OSSkuName	OS edition friendly name (currently Windows only)	Categorical
OSInstallTypeName	Friendly description of what install was used on the machine	Categorical
OSInstallLanguageIdentifier	NA	Categorical
OSUILocaleIdentifier	NA	Categorical
OSWUAutoUpdateOptionsName	Friendly name of the WindowsUpdate auto	Categorical
IsPortableOperatingSystem	Indicates whether OS is booted up and running via Windows	Binary
GenuineStateName	Friendly name of OSGenuineStateID. 0 = Genuine	Categorical
ActivationChannel	Retail license key or Volume license key for a machine.	Categorical
IsFlightingInternal	NA	Binary
IsFlightsDisabled	Indicates if the machine is participating in flighting.	Binary
FlightRing	The ring that the device user would like to receive flights for.	Categorical
ThresholdOptIn	NA	Binary
FirmwareManufacturerIdentifier	NA	Categorical
FirmwareVersionIdentifier	NA	Categorical
IsSecureBootEnabled	Indicates if Secure Boot mode is enabled.	Binary
IsWIMBootEnabled	NA	Binary
IsVirtualDevice	Identifies a Virtual Machine (machine learning model)	Binary
IsTouchEnabled	Is this a touch device ?	Binary
IsPenCapable	Is the device capable of pen input ?	Binary
IsAlwaysOnAlwaysConnected	Whether the battery enables device to be AlwaysConnected .	Binary
Wdft_IsGamer	Whether the device is a gamer device or not based on its HW.	Binary
Wdft_RegionIdentifier	NA	Categorical

8 Appendix: Code

8.1 Data Loading

Here I do not include the initial data clean-up for it is extremely long and uninteresting.

```
# Import libraries
from pyspark.sql.functions import isnan, when, count, col
from pyspark.sql.types import DoubleType, StringType, IntegerType
from pyspark.sql import SparkSession
import numpy as np

# Initialize Spark session
spark = SparkSession.builder.appName('cleanup').getOrCreate()

# Import the data into a Spark DataFrame with the schema
file_location = "/user/mie_sbetancourt/PROJECT/Data/data_reduced_reweighted_FINAL_3.csv"
data = spark.read.format("csv").option("header","true").option("inferSchema","true").load(file_location)

# Last-minute clean-up
data = (data.drop("Census_FirmwareVersionIdentifier")
        .withColumn("AVProductsEnabled", when(data["AVProductsEnabled"]=="unknown", -1).otherwise(data["AVPro

data_1 = (data.withColumn("AVProductsEnabled", data["AVProductsEnabled"].cast(IntegerType()))
          .withColumn("AvSigVersion_new", data["AvSigVersion_new"].cast(StringType()))
          .withColumn("Census_OSBuildNumber", data["Census_OSBuildNumber"].cast(StringType()))
          .withColumn("Census_OSBuildRevision", data["Census_OSBuildRevision"].cast(StringType()))
          .withColumn("Census_OSUILocaleIdentifier", data["Census_OSUILocaleIdentifier"].cast(StringType()))
          .withColumn("Census_OSVersion_new", data["Census_OSVersion_new"].cast(StringType()))
          .withColumn("CountryIdentifier", data["CountryIdentifier"].cast(StringType()))
          .withColumn("LocaleEnglishNameIdentifier", data["LocaleEnglishNameIdentifier"].cast(StringType()))
          .withColumn("OsBuild", data["OsBuild"].cast(StringType()))
          .withColumn("OsSuite", data["OsSuite"].cast(StringType())))
data_1 = data_1.withColumnRenamed("HasDetections","label").drop("OsBuildLab_new")

stringCols = []
for col in data_1.dtypes:
    if col[1] == 'string':
        stringCols.append(col[0])
stringCols.remove("MachineIdentifier")

numericCols = np.setdiff1d(data_1.columns, stringCols).tolist()
numericCols.remove("MachineIdentifier")
numericCols.remove("classWeightCol")
numericCols.remove("label")

# Setting random seed for reproducibility
sampling_seed=1111

trainingData1 = data_1.sampleBy("label", fractions={0: .1, 1: 1}, seed=sampling_seed)
trainingData = trainingData1.sampleBy("label", fractions={0: .9, 1: .9}, seed=sampling_seed)
# Subtracting 'train' from original 'data' to get test set
testData = trainingData1.subtract(trainingData)
```

8.2 Logistic Regression (LR)

```
from pyspark.conf import SparkConf
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Spark_LR").getOrCreate()
spark.sparkContext._conf.getAll()

conf = spark.sparkContext._conf.setAll([('spark.executor.memory', '95g'),
                                        ('spark.app.name', 'Spark_GBM'),
                                        ('spark.executor.cores', '4'),
                                        ('spark.cores.max', '4'),
                                        ('spark.driver.memory','95g')])

#Stop the current Spark Session
spark.sparkContext.stop()

#Create a Spark Session
spark = SparkSession.builder.config(conf=conf).getOrCreate()

# Import libraries
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

import time
start_time = time.time()

sampling_seed=1111
# The index of string values multiple columns
indexers = [
    StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c),handleInvalid="skip")
    for c in stringCols
]

# The encode of indexed vlaues multiple columns
encoders = [OneHotEncoder(dropLast=False,inputCol=indexer.getOutputCol(),
            outputCol="{0}_encoded".format(indexer.getOutputCol()))
    for indexer in indexers
]

lr = (LogisticRegression(labelCol="label", featuresCol="features"))
      #,weightCol="classWeightCol")) maxIter=100)) #, regParam=0.1, elasticNetParam=0.5))

# Vectorizing encoded values
assembler = VectorAssembler(inputCols=([encoder.getOutputCol() for encoder in encoders] + numericCols),
                            outputCol="features")

pipeline = Pipeline(stages=indexers + encoders+[assembler]+[lr])

estimatorParam = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.0, .025, 1.0]) \
    .addGrid(lr.elasticNetParam, [0.0, .5, 1.0])  \
    .build()

evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction")
```

```
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=estimatorParam,
                          evaluator=evaluator,
                          numFolds=3,
                          parallelism=8,
                          seed=sampling_seed)

cvmodel = crossval.fit(trainingData)
# Note that the evaluator here is a BinaryClassificationEvaluator and its default metric
# is areaUnderROC.

#Saving the trained model
lr_path = temp_path + "/lr"
lr.save(lr_path)
model_path = temp_path + "/lr_model"
lr_cvmodel.bestModel.save(model_path)

print("--- %s seconds ---" % (time.time() - start_time))


###Loading model
from pyspark.ml import *
lr_model_path = temp_path + "/lr_model"
lr_mod2 = PipelineModel.load(lr_model_path)

##Evaluating Performance Metrics
lr_predictions = lr_mod2.transform(testData)
lr_train_predictions = lr_mod2.transform(trainingData)
print("The area under ROC for train set after CV  is {}".format(evaluator.evaluate(lr_train_predictions)))
print("The area under ROC for test set after CV  is {}".format(evaluator.evaluate(lr_predictions)))
print('Best regParam: ', lr_mod2.stages[-1]._java_obj.getRegParam())
print('Best elasticNetParam: ', lr_mod2.stages[-1]._java_obj.getElasticNetParam())

#plotting the ROC Curve
trainingSummary = lr_mod2.stages[-1].summary
roc = trainingSummary.roc.toPandas()
plt.plot([0,1],'r--')
plt.plot(roc['FPR'],roc['TPR'])
plt.ylabel('False Positive Rate')
plt.xlabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```

8.3 Support Vector Machine (SVM)

```
from pyspark.conf import SparkConf
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Spark_SVM").getOrCreate()
spark.sparkContext._conf.getAll()

conf = spark.sparkContext._conf.setAll([('spark.executor.memory', '95g'),
                                        ('spark.app.name', 'Spark_GBM'),
                                        ('spark.executor.cores', '4'),
                                        ('spark.cores.max', '4'),
                                        ('spark.driver.memory','95g')])

#Stop the current Spark Session
```

```
spark.sparkContext.stop()

#Create a Spark Session
spark = SparkSession.builder.config(conf=conf).getOrCreate()

# Import libraries
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.classification import LinearSVC
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

temp_path = "/user/mie_sbetancourt/PROJECT/"

import time
start_time = time.time()

sampling_seed=1111
# The index of string values multiple columns
indexers = [
    StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c),handleInvalid="skip")
    for c in stringCols
]

# The encode of indexed vlaues multiple columns
encoders = [OneHotEncoder(dropLast=False,inputCol=indexer.getOutputCol(),
            outputCol="{0}_encoded".format(indexer.getOutputCol()))
    for indexer in indexers
]

lsvc = (LinearSVC(labelCol="label", featuresCol="features"))
      #,weightCol="classWeightCol")) maxIter=100)) #, regParam=0.1, elasticNetParam=0.5))

# Vectorizing encoded values
assembler = VectorAssembler(inputCols=([encoder.getOutputCol() for encoder in encoders] + numericCols),
                            outputCol="features")

pipeline = Pipeline(stages=indexers + encoders+[assembler]+[lsvc])

estimatorParam = ParamGridBuilder() \
    .addGrid(lsvc.regParam, [.025, .01, .05]) \
    .addGrid(lsvc.fitIntercept, [True, False])  \
    .addGrid(lsvc.standardization, [True, False])  \
    .build()

evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction")

crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=estimatorParam,
                          evaluator=evaluator,
                          numFolds=3,
                          parallelism=8,
                          seed=sampling_seed)

lsvc_cvmodel = crossval.fit(trainingData)
# Note that the evaluator here is a BinaryClassificationEvaluator and its default metric
# is areaUnderROC.

#Saving trained model
```

```
lsvc_path = temp_path + "/lsvc"
lsvc.save(lsvc_path)
model_path = temp_path + "/lsvc_model"
lsvc_cvmodel.bestModel.save(model_path)


print("--- %s seconds ---" % (time.time() - start_time))


###Loading model
from pyspark.ml import *
lsvc_model_path = temp_path + "/lsvc_model"
lsvc_mod2 = PipelineModel.load(lsvc_model_path)

##Evaluating Performance Metrics
lsvc_predictions = lsvc_mod2.transform(testData)
lsvc_train_predictions = lsvc_mod2.transform(trainingData)
print("The area under ROC for train set after CV  is {}".format(evaluator.evaluate(lsvc_train_predictions)))
print("The area under ROC for test set after CV  is {}".format(evaluator.evaluate(lsvc_predictions)))
print('Best regParam: ', lsvc_mod2.stages[-1]._java_obj.getRegParam())
print('Best fitIntercept: ', lsvc_mod2.stages[-1]._java_obj.getFitIntercept())
```

8.4 Random Forest (RF)

```
from pyspark.conf import SparkConf
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Spark_RF").getOrCreate()
spark.sparkContext._conf.getAll()

conf = spark.sparkContext._conf.setAll([('spark.executor.memory', '95g'),
                                        ('spark.app.name', 'Spark_GBM'),
                                        ('spark.executor.cores', '4'),
                                        ('spark.cores.max', '4'),
                                        ('spark.driver.memory','95g')])

#Stop the current Spark Session
spark.sparkContext.stop()

#Create a Spark Session
spark = SparkSession.builder.config(conf=conf).getOrCreate()

#MODEL
temp_path = "/user/mie_sbetancourt/PROJECT/"


from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
#from FeatureImportanceSelector import ExtractFeatureImp, FeatureImpSelector

import time
start_time = time.time()

sampling_seed=1111
```

```
# The index of string values multiple columns
indexers = [
    StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c),handleInvalid="keep")
    for c in stringCols
]

# The encode of indexed vlaues multiple columns
encoders = [OneHotEncoder(dropLast=False,inputCol=indexer.getOutputCol(),
            outputCol="{0}_encoded".format(indexer.getOutputCol()))
    for indexer in indexers
]

randfor = (RandomForestClassifier(labelCol="label", featuresCol="features"))

# Vectorizing encoded values
assembler = VectorAssembler(inputCols=([encoder.getOutputCol() for encoder in encoders] + numericCols),
                            outputCol="features")

pipeline = Pipeline(stages=indexers + encoders+[assembler]+[randfor])

#Hyperparam grid
estimatorParam = ParamGridBuilder() \
    .addGrid(randfor.numTrees, [100, 125,175]) \
    .addGrid(randfor.maxDepth, [12,15]) \
    .addGrid(randfor.maxBins, [25]) \
    .build()

evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction")

crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=estimatorParam,
                          evaluator=evaluator,
                          numFolds=3,
                          parallelism=8,
                          seed=sampling_seed)

rand_cvmodel = crossval.fit(trainingData)
# Note that the evaluator here is a BinaryClassificationEvaluator and its default metric
# is areaUnderROC.

#Saving trained model
rand_path = temp_path + "/rand"
randfor.save(rand_path)
model_path = temp_path + "/rand_model"
rand_cvmodel.bestModel.save(model_path)


print("--- %s seconds ---" % (time.time() - start_time))

#Loading the model and calculating metrics
from pyspark.ml import *
rand_model_path = temp_path + "/rand_model"
rand_mod2 = PipelineModel.load(rand_model_path)
rand_predictions = rand_mod2.transform(testData)
rand_train_predictions = rand_mod2.transform(trainingData)
print("The area under ROC for train set is {}".format(evaluator.evaluate(rand_train_predictions)))
print("The area under ROC for test set is {}".format(evaluator.evaluate(rand_predictions)))
print ('Best numTrees: ', rand_mod2.stages[-1]._java_obj.getMaxDepth()
print ('Best maxBins: ', rand_mod2.stages[-1]._java_obj.getMaxBins())
```

```
print ('Best maxDepth: ', rand_mod2.stages[-1]._java_obj.getMaxDepth())
```

8.5 Gradient-Boosted Tress (GBT)

```
from pyspark.conf import SparkConf
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Spark_GBM").getOrCreate()
spark.sparkContext._conf.getAll()

conf = spark.sparkContext._conf.setAll([('spark.executor.memory', '95g'),
                                        ('spark.app.name', 'Spark_GBM'),
                                        ('spark.executor.cores', '4'),
                                        ('spark.cores.max', '4'),
                                        ('spark.driver.memory','95g')])

#Stop the current Spark Session
spark.sparkContext.stop()

#Create a Spark Session
spark = SparkSession.builder.config(conf=conf).getOrCreate()

#MODEL
temp_path = "/user/mie_sbetancourt/PROJECT/"

from pyspark.ml.classification import GBTClassifier
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

import time
start_time = time.time()

sampling_seed=1111

# The index of string values multiple columns
indexers = [
    StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c),handleInvalid="keep")
    for c in stringCols
]

# The encode of indexed vlaues multiple columns
encoders = [OneHotEncoder(dropLast=False,inputCol=indexer.getOutputCol(),
            outputCol="{0}_encoded".format(indexer.getOutputCol()))
    for indexer in indexers
]

gbt = (GBTClassifier(labelCol="label", featuresCol="features"))

# Vectorizing encoded values
assembler = VectorAssembler(inputCols=([encoder.getOutputCol() for encoder in encoders] + numericCols),
                            outputCol="features")

pipeline = Pipeline(stages=indexers + encoders+[assembler]+[gbt])

estimatorParam = ParamGridBuilder() \
    .addGrid(gbt.maxDepth, [4, 6, 8, 10]) \
```

```
    .addGrid(gbt.maxBins, [15, 25])  \
    .addGrid(gbt.stepSize, [0.1, 0.05]) \
    .addGrid(gbt.subsamplingRate, [.7]) \
    .build()

evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction")

gbt_crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=estimatorParam,
                          evaluator=evaluator,
                          numFolds=3,
                          parallelism=8,
                          seed=sampling_seed)

gbt_cvmodel = gbt_crossval.fit(trainingData)
# Note that the evaluator here is a BinaryClassificationEvaluator and its default metric
# is areaUnderROC.

#Saving trained model
gbt_path = temp_path + "/gbt"
gbt.save(gbt_path)
model_path = temp_path + "/gbt_model"
gbt_cvmodel.bestModel.save(model_path)


print("--- %s seconds ---" % (time.time() - start_time))

#Fetching trained model and printing out metrics
from pyspark.ml import *
gbt_model_path = temp_path + "/gbt_model"
gbt_mod2 = PipelineModel.load(gbt_model_path)
gbt_predictions = gbt_mod2.transform(testData)
gbt_train_predictions = gbt_mod2.transform(trainingData)
print("The area under ROC for train set after CV  is {}".format(evaluator.evaluate(gbt_train_predictions)))
print("The area under ROC for test set after CV  is {}".format(evaluator.evaluate(gbt_predictions)))
print ('Best maxDepth: ', gbt_mod2.stages[-1]._java_obj.getMaxDepth()
print ('Best maxBins: ', gbt_mod2.stages[-1]._java_obj.getMaxBins())
print('Best stepSize: ', gbt_mod2.stages[-1]._java_obj.getStepSize())
```

9 Appendix: Imbalanced Data (Extra)

9.1 Stratified Cross-Validator Function

```
import itertools
import numpy as np

from pyspark import since, keyword_only
from pyspark.ml import Estimator, Model
from pyspark.ml.common import _py2java
from pyspark.ml.param import Params, Param, TypeConverters
from pyspark.ml.param.shared import HasSeed
from pyspark.ml.tuning import CrossValidator, CrossValidatorModel
from pyspark.ml.util import *
from pyspark.ml.wrapper import JavaParams
from pyspark.sql.functions import rand
from functools import reduce

class StratifiedCrossValidator(CrossValidator):
  def stratify_data(self, dataset):
    """
    Returns an array of dataframes with the same ratio of passes and failures.
    Currently only supports binary classification problems.
    """

    epm = self.getOrDefault(self.estimatorParamMaps)
    numModels = len(epm)
    nFolds = self.getOrDefault(self.numFolds)
    split_ratio = 1.0 / nFolds

    passes = dataset[dataset['label'] == 1]
    fails = dataset[dataset['label'] == 0]

    pass_splits = passes.randomSplit([split_ratio for i in range(nFolds)])
    fail_splits = fails.randomSplit([split_ratio for i in range(nFolds)])

    stratified_data = [pass_splits[i].unionAll(fail_splits[i]) for i in range(nFolds)]

    return stratified_data

  def _fit(self, dataset):
    est = self.getOrDefault(self.estimator)
    epm = self.getOrDefault(self.estimatorParamMaps)
    numModels = len(epm)
    eva = self.getOrDefault(self.evaluator)
    nFolds = self.getOrDefault(self.numFolds)
    seed = self.getOrDefault(self.seed)
    metrics = [0.0] * numModels

    stratified_data = self.stratify_data(dataset)

    for i in range(nFolds):
      train_arr = [x for j,x in enumerate(stratified_data) if j != i]
      train = reduce((lambda x, y: x.unionAll(y)), train_arr)
      validation = stratified_data[i]

      models = est.fit(train, epm)
```

```
      for j in range(numModels):
        model = models[j]
        metric = eva.evaluate(model.transform(validation, epm[j]))
        metrics[j] += metric/nFolds

    if eva.isLargerBetter():
      bestIndex = np.argmax(metrics)
    else:
      bestIndex = np.argmin(metrics)

    bestModel = est.fit(dataset, epm[bestIndex])
    return self._copyValues(CrossValidatorModel(bestModel, metrics))
```

9.2 Logistic Regression with Stratified Cross-validation and Rebalancing

```
from pyspark.conf import SparkConf
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Spark_Imbalanced_LR").getOrCreate()
spark.sparkContext._conf.getAll()

conf = spark.sparkContext._conf.setAll([('spark.executor.memory', '95g'),
                                        ('spark.app.name', 'Spark_GBM'),
                                        ('spark.executor.cores', '4'),
                                        ('spark.cores.max', '4'),
                                        ('spark.driver.memory','95g')])

#Stop the current Spark Session
spark.sparkContext.stop()

#Create a Spark Session
spark = SparkSession.builder.config(conf=conf).getOrCreate()

# Import libraries
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

import time
start_time = time.time()

sampling_seed=1111

# The index of string values multiple columns
indexers = [
    StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c),handleInvalid="skip")
    for c in stringCols
]

# The encode of indexed vlaues multiple columns
encoders = [OneHotEncoder(dropLast=False,inputCol=indexer.getOutputCol(),
            outputCol="{0}_encoded".format(indexer.getOutputCol()))
    for indexer in indexers
]

ulr = (LogisticRegression(labelCol="label",
```

```
                        featuresCol="features",weightCol="classWeightCol"))

# Vectorizing encoded values
assembler = VectorAssembler(inputCols=([encoder.getOutputCol() for encoder in encoders] + numericCols),
                            outputCol="features")

pipeline = Pipeline(stages=indexers + encoders+[assembler]+[ulr])

estimatorParam = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.0, .025, .1, 1.0]) \
    .addGrid(lr.elasticNetParam, [0.0, .2,.8, 1.0])  \
    .addGrid(lr.fitIntercept,[True,False])  \
    .build()

evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction",
                                          metricName="areaUnderPR")

crossval = StratifiedCrossValidator(estimator=pipeline,
                          estimatorParamMaps=estimatorParam,
                          evaluator=evaluator,
                          numFolds=3,
                          parallelism=8,
                          seed=sampling_seed)

ulr_cvmodel = crossval.fit(trainingData)
# Note that the evaluator here is a BinaryClassificationEvaluator and its default metric
# is areaUnderROC.

print("--- %s seconds ---" % (time.time() - start_time))



ulr_path = temp_path + "/ulr"
ulr.save(ulr_path)
model_path = temp_path + "/ulr_model"
ulr_cvmodel.bestModel.save(model_path)


from pyspark.ml import *
ulr_model_path = temp_path + "/ulr_model"
ulr_mod2 = PipelineModel.load(ulr_model_path)
ulr_predictions = ulr_mod2.transform(testData)
ulr_train_predictions = ulr_mod2.transform(trainingData)
print("The area under PR for train set is {}".format(evaluator.evaluate(ulr_train_predictions)))
print("The area under PR for test set is {}".format(evaluator.evaluate(ulr_predictions)))
print ('Best elasticNetParam: ', ulr_mod2.stages[-1]._java_obj.getElasticNetParam())
print ('Best regParam: ', ulr_mod2.stages[-1]._java_obj.getRegParam())
```