Machine Learning Case Study With Pyspark

0. Some random thoughts/babbling

I guess it is the best time, since you can deal with millions of data points with relatively limited computing power, and without having to know every single bit of computer science. They called it high-level. It is also the worst time, since like the wild west, there is all kinds of tools hurling in front you: from ancient dinosaurs like SAS, to the modern helicopter Apache Spark. No pun intended.

I got to use spark at the 2.0 era, the era of DataFrame. I love it since I am familiar with pandas and numpy already. Even better, I can convert DataFrame into pandas if I cannot figure out how to do it in the spark way.

Well, how fast it can be? It takes me half a hour to finish a full machine learning process, from imputation and one hot encoding stuff, and ends with a random forest. By the way, the data set has got 7 million data points with 68 variables. And I only used the community free version from databricks. Kudos to them.

1. Background Information

This is a classification problem, where the response variable has two categories. And I am going to first demonstrate a minimal amount of a complete workflow. However, two things might be considered in addition:

1. It is always a good idea to some exploratory work in the first place. And in my opinion, it might be sensible idea to do this in `numpy` and `pandas` (a.k.a. base python) in a smaller data set. I have another article to talk about this specifically.

2. More fine tuning work is rewarding. And ensemble and stacking should also be considered, along with other subtleties. Find my article he

2. Data Preprocessing

```
df = spark\
       .read.format("com.databricks.spark.csv")\
       .option("header", "true")\
       .option("inferschema", "true")\
       .load("/FileStore/tables/cazka1531500053994895/example.csv")

  from collections import defaultdict

  data_types = defaultdict(list)
  for entry in df.schema.fields:
    data_types[str(entry.dataType)].append(entry.name)
```

The first couple lines loads the data and creates a data frame object. `inferschema` is `true` can give a good guess about the data type for each column. And I created a dictionary to store them.. In this case, we got string type, double type and integer type.

This might come as a surprise, but the difference between integer type and double type is important in Spark. For instance, there is a new function called `Imputer` in Spark 2.2, which can only work with double type, and will throw an error if you pass in an integer variable. If you do not care about it, just cast integer type to double.

2.1 Handling categorical data

Let's first deal with the string types. Namely, deleting the variable with too many categories, and handling missing data.

```
from pyspark.sql.functions import countDistinct, approxCountDistinct
  counts_summary = df.agg(*[countDistinct(c).alias(c) for c in data_types['
  counts_summary = counts_summary.toPandas()

  import pandas as pd
  counts = pd.Series(counts_summary.values.ravel())
  counts.index = counts_summary.columns
```

```
    counts.index = counts_summary.columns

    sorted_vars = counts.sort_values(ascending = False)
    ignore = list((sorted_vars[sorted_vars >100]).index)
```

You can use `approxCountDistinct` instead of `countDistinct` since the latter is more expensive computationally. The above couple lines finds the categorical variables with more than 100 categories. This kind of variables, like IDs, usually are not informative. Of course, if you feel uncomfortable in ditching it, we'll talk about another way to deal with it. For now, we simply ditch all variables in `ignore`.

Secondly, for other categorical variables that remains in the model, we need to deal with missing data. The strategy is to create a new category called `missing` to store them. Here is the implementation.

```
  strings_used = [var for var in data_types["StringType"] if var not in ignor

    missing_data_fill = {}
    for var in strings_used:
      missing_data_fill[var] = "missing"



    df = df.fillna(missing_data_fill)
```

`string_used` is a list with all string type variables excluding the ones with more than 100 categories. We next pass a dictionary to `fillna` in order to replace all `NA` witsth the string `missing`.

However, computers are never designed to deal with strings and texts. We need to convert the categorical variables into numbers. The process includes string indexing and one hot encoding. For example, translating "man" and "women" into 1 and 0 is string indexing, and one-hot encoding is a little more complex. Here is one example.

You have three categories. Let's say republican, democrat and other. After string indexing we end up with 0, 1, 2. To one-hot encode them, you can only display them with 0 and 1 like this.

republican	democrat	Other
1	0	0
0	1	0
0	0	1

By the way, one-hot is an electric engineering terms, which means you can literally only fire up a semiconductor one at a time. In fact, you do not have to understand what happens under the hood since Spark provides the `StringIndexer` and `OneHotEncoder` in the `ml` library.

```
from pyspark.ml import Pipeline
  from pyspark.ml.feature import OneHotEncoder, StringIndexer

  stage_string = [StringIndexer(inputCol= c, outputCol= c+"_string_encoded'
  stage_one_hot = [OneHotEncoder(inputCol= c+"_string_encoded", outputCol=

  ppl = Pipeline(stages= stage_string + stage_one_hot)
  df = ppl.fit(df).transform(df)
```

Two things. First, the `OneHotEncoder` and `StringIndexer` , unlike `Imputer` , which is something we will talk about later, can only take one column at a time, and thus we need a list comprehension and a pipeline.

A pipeline is an interesting idea first emerges in the `sklearn` library, in which you can feed a series of tasks you want to do, and make them a list and the pipeline will handle everything for you. This is life saving because if you do manually, you'll have to do imputation, and save the data, and pass it on to the next task, and repeat this cycle again and again.

Another thing is, this pipeline object is slightly a little different than the one in `sklearn` . In `sklearn` , you do not need to do `model_fit = model.fit(data)` like we have here. You only have to do `model.fit(data)` and the updated information is already in the `model` object and we do not need to pass it on to another variable.

2.2 Handling numerical data

In handling numerical data, I am going to demonstrate two different approaches. In Spark 2.2, a new `Imputer` function is included, but only . .\ for double type and not for integers. This is how it works.

```
numericals = data_types["DoubleType"]
  numericals = [var for var in numericals if var not in ignore]
  numericals_imputed = [var + "_imputed" for var in numericals]

  from pyspark.ml.feature import Imputer

  imputer = Imputer(inputCols = numericals, outputCols = numericals_imputec
  df = imputer.fit(df).transform(df)
```

Note that the new function can handle multiple columns at one time. I guess this is where Spark is headed to since handling multiple variables at a time is a much more common scenario than one column at a time. Obviously the imputed columns all end with `_imputed`.

What about integer type? Two different strategies. One, cast integer to double type and use the `Imputer` function. Two, use the old way before Spark 2.2: pass a dictionary to fill `NA` with `dropna` function. We typically fill `NA` with sample mean.

Here is the implementation of the first way.

```
for c in data_types["IntegerType"]:
    df = df.withColumn(c+ "_cast_to_double", df[c].cast("double"))

  cast_vars = [var for var in  df.columns if var.endswith("_cast_to_double'
  cast_vars_imputed  = [var+ "imputed" for var in cast_vars]

  imputer_for_cast_vars = Imputer(inputCols = cast_vars, outputCols = cast_
  df = imputer_for_cast_vars.fit(df).transform(df)
```

Note that `withColumn` is the most common way to add a new column, where the first argument being name of the new column and the second argument is the operation. You can define your own operation by `udf` as well.

Here is the second strategy, and let's pretend there is no `Imputer` fur whatsoever.

```
mean_dict = { col: 'mean' for col in numericals }
  col_avgs = only_nums.agg( mean_dict ).collect()[0].asDict()
  col_avgs = { k[4:-1]: v for k,v in col_avgs.iteritems()}
  only_nums = only_nums.fillna( col_avgs )
```

For the `agg` function, we can pass in a dictionary like `{"column1": mean, "column2: max}` , in which the key is column name and the value is the operation for that column. And thus `col_avgs` is a dictionary with column names and column mean, which is later feed into `fillna` method.

2.3 Put them together

The idea here is to assemble everything into a vector. This is reasonable since after one-hot encoding and stuff, you end up with a mishmash of integers, floats, sparse vectors, and maybe dense vectors. And what we do next is bundle them altogether and call it features.

```
from pyspark.ml.feature import VectorAssembler

features = cast_vars_imputed + numericals_imputed \
            + [var + "_one_hot" for var in strings_used]

vector_assembler = VectorAssembler(inputCols = features, outputCol= "feat
data_training_and_test = vector_assembler.transform(df)
```

Interestingly, if you do not specify any variables for the algorithm to look at, the `sklearn` library will throw an error, but Spark will first look for `features` as X, and `labels` in default. That's why we usually call the assembled features `features` .

3. Dimension Reduction/feature selection

The most popular way in doing dimension reduction is principal component analysis, a.k.a PCA.

The dimension reduction intends to project the variables into a lower dimension space space. An additional benefit is these dimensions are usually independent with each other. Hence, one may find it helpful when dealing with lots of highly correlated variables. We can choose the dimension `k` in the `PCA()` method. We can find the best `k` by cross validation.

For instance, if we want to extract 30 features from 68, the following code can be used.

```
from pyspark.ml.feature import PCA

  pca_model = PCA(k = 30,inputCol = "features", outputCol = "pca_features")
  model = pca_model.fit(data_training_and_test)
  data_training_and_test = model.transform(data_training_and_test)
```

4. Modeling fitting and tuning

We got couple of built-in classifiers, including random forest, boosting trees, logistic regression and etc. We will implement random forest as an example, and the only parameter one needs to specify is the number of trees in the classifier.

```
from pyspark.ml.classification import RandomForestClassifier

  (training_data, test_data) = data_training_and_test.randomSplit([0.7, 0.3
  rf = RandomForestClassifier(labelCol = "labels", featuresCol = "pca_featu
  rf_model = rf.fit(training_data)
```

Remember that we arbitrarily used `k = 30` during the PCA, and here `numTrees = 20`. To find the best parameters, one should consider grid search. Luc
we do not need to code all the grid search and cross-validation thing ourselves. Here is how to do it with the `ml` library.

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
  from pyspark.ml.evaluation import BinaryClassificationEvaluator as BCE

  very_small_sample = data_training_and_test.sample(False, 0.001).cache()

  pca_model = PCA(inputCol = "features", outputCol = "pca_features_cv")
  rf = RandomForestClassifier(labelCol = "labels", featuresCol = "pca_featu
  ppl_cv = Pipeline(stages = [pca_model, rf])

  paramGrid = ParamGridBuilder() \
      .addGrid(pca_model.k, [10, 20, 30, 40 50]) \
      .addGrid(rf.numTrees, [20, 30, 50]) \
      .build()

  crossval = CrossValidator(estimator = ppl_cv,
                            estimatorParamMaps=paramGrid,
                            evaluator = BCE(labelCol = "STP_UP_IND",\
                                            rawPredictionCol = "probability
                                            metricName = "areaUnderROC"),
                            numFolds= 3,
                            metricName = "accuracy")

  cv_model = crossval.fit(very_small_sample)
```

Note that we got 2 grid, and got 3 x 5 different combinations to compute. No matter how powerful your server is, I still think we should do it on a smaller subset of the original file. Be reasonable.

You can choose different `metricName` to match your purpose. And in this case we use area under the ROC curve as the criteria.

```
predictions = cv_model.transform(test_data)
  evaluator= BinaryClassificationEvaluator(labelCol = "STP_UP_IND", rawPred
  accuracy = evaluator.evaluate(predictions)
```

The accuracy score for this model is 0.54, which is an OK model. But how to make it better? Let's find out in the next article.