Feature Selection Using Feature Importance Score - Creating a PySpark Estimator

Posted on June 19, 2018

In this post I discuss how to create a new pyspark estimator to integrate in an existing machine learning pipeline. This is an extension of my previous post (https://www.timlrx.com/2018/04/08/creating-a-custom-cross-validation-function-in-pyspark/) where I discussed how to create a custom cross validation function. Recently, I have been looking at integrating existing code in the pyspark ML pipeline framework. A pipeline is a fantastic concept of abstraction since it allows the analyst to focus on the main tasks that needs to be carried out and allows the entire piece of work to be reusable.

As a fun and useful example, I will show how feature selection using feature importance score can be coded into a pipeline. I find Pyspark's MLlib native feature selection functions relatively limited so this is also part of an effort to extend the feature selection methods. Here, I use the feature importance score as estimated from a model (decision tree / random forest / gradient boosted trees) to extract the variables that are plausibly the most important.

First, let's setup the jupyter notebook and import the relevant functions. I use a local version of spark to illustrate how this works but one can easily use a yarn cluster instead.

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import numpy as np
import pandas as pd
pd.options.display.max_columns = None

import findspark
findspark.init()
from pyspark import SparkContext
from pyspark import SQLContext
```

```
sc = SparkContext()
spark = SQLContext(sc)
```

```
from pyspark.sql.functions import *
from pyspark.ml.classification import  RandomForestClassifier
from pyspark.ml.feature import StringIndexer, OneHotEncoderEstimator, VectorAssembler, VectorSlicer
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.linalg import Vectors
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
```

Bank Marketing Data Set

To show the usefulness of feature selection and to sort of validate the script, I used the Bank Marketing Data Set from UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/bank+marketing) as an example throughout this post. This comes from Moro et al., 2014 paper on A Data-Driven Approach to Predict the Success of Bank Telemarketing. As the name of the paper suggests, the goal of this dataset is to predict which bank customers would subscribe to a term deposit product as a result of a phone marketing campaign.

Let us read in the file and take a look at the variables of the dataset.

```
ad.option("delimiter", ";").csv("../data/bank-additional/bank-additional-full.csv", header=True, inferSchema = True)
```

```
df.dtypes
```

```
[('age', 'int'),
 ('job', 'string'),
 ('marital', 'string'),
 ('education', 'string'),
 ('default', 'string'),
 ('housing', 'string'),
 ('loan', 'string'),
 ('contact', 'string'),
 ('month', 'string'),
 ('day_of_week', 'string'),
 ('duration', 'int'),
 ('campaign', 'int'),
 ('pdays', 'int'),
 ('previous', 'int'),
 ('poutcome', 'string'),
 ('emp.var.rate', 'double'),
 ('cons.price.idx', 'double'),
 ('cons.conf.idx', 'double'),
 ('euribor3m', 'double'),
 ('nr.employed', 'double'),
 ('y', 'string')]
```

There are some problematic variable names and we should replace the dot seperator with an underscore.

```
df = df.toDF(*(c.replace('.', '_') for c in df.columns))
```

```
df.limit(5).toPandas()
```

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	2
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	1
2	37	services	married	high.school	no	yes	no	telephone	may	mon	2
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	1
4	56	services	married	high.school	no	no	yes	telephone	may	mon	3

It's always nice to take a look at the distribution of the variables

```
df.describe().toPandas()
```

	summary	age	job	marital	education	default	housing	loan	contact	m
0	count	41188	41188	41188	41188	41188	41188	41188	41188	4
1	mean	40.02406040594348	None	None	None	None	None	None	None	N
2	stddev	10.421249980934057	None	None	None	None	None	None	None	N
3	min	17	admin.	divorced	basic.4y	no	no	no	cellular	a
4	max	98	unknown	unknown	unknown	yes	yes	yes	telephone	s

There are quite a few variables that are encoded as a string in this dataset. Converting strings to a binary indicator variable / dummy variable takes up quite a few degrees of freedom. In machine learning speak it might also lead to the model being overfitted. Let us take a look at what is represented by each variable that is of string type.

```
for i in df.dtypes:
    if i[1]=='string':
        df.groupBy(i[0]).count().orderBy('count', ascending=False).toPandas()
```

	job	count
0	admin.	10422
1	blue-collar	9254
2	technician	6743
3	services	3969
4	management	2924
5	retired	1720
6	entrepreneur	1456
7	self-employed	1421
8	housemaid	1060
9	unemployed	1014
10	student	875
11	unknown	330

	marital	count
0	married	24928
1	single	11568
2	divorced	4612
3	unknown	80

	education	count
0	university.degree	12168
1	high.school	9515
2	basic.9y	6045
3	professional.course	5243
4	basic.4y	4176
5	basic.6y	2292
6	unknown	1731
7	illiterate	18

	default	count
0	no	32588
1	unknown	8597
2	yes	3

	housing	count

	housing	count
0	yes	21576
1	no	18622
2	unknown	990

	loan	count
0	no	33950
1	yes	6248
2	unknown	990

	contact	count
0	cellular	26144
1	telephone	15044

	month	count
0	may	13769
1	jul	7174
2	aug	6178
3	jun	5318
4	nov	4101
5	apr	2632
6	oct	718
7	sep	570
8	mar	546
9	dec	182

	day_of_week	count
0	thu	8623
1	mon	8514
2	wed	8134
3	tue	8090
4	fri	7827

	poutcome	count
0	nonexistent	35563
1	failure	4252
2	success	1373

	y	count
0	no	36548
1	yes	4640

The number of categories for each string type is relatively small which makes creating binary indicator variables / one-hot encoding a suitable pre-processing step. Let us take a look at how to do feature selection using the feature importance score the manual way before coding it as an estimator to fit into a Pyspark pipeline.

Data Preprocessing

Before we run the model on the most relevant features, we would first need to encode the string variables as binary vectors and run a random forest model on the whole feature set to get the feature importance score. Here I just run most of these tasks as part of a pipeline.

```
# one hot encoding and assembling
encoding_var = [i[0] for i in df.dtypes if (i[1]=='string') & (i[0]!='y')]
num_var = [i[0] for i in df.dtypes if ((i[1]=='int') | (i[1]=='double')) & (i[0]!='y')]

string_indexes = [StringIndexer(inputCol = c, outputCol = 'IDX_' + c, handleInvalid = 'keep') for c in encoding_var]
onehot_indexes = [OneHotEncoderEstimator(inputCols = ['IDX_' + c], outputCols = ['OHE_' + c]) for c in encoding_var]
label_indexes = StringIndexer(inputCol = 'y', outputCol = 'label', handleInvalid = 'keep')
assembler = VectorAssembler(inputCols = num_var + ['OHE_' + c for c in encoding_var], outputCol = "features")
rf = RandomForestClassifier(labelCol="label", featuresCol="features", seed = 8464,
                            numTrees=10, cacheNodeIds = True, subsamplingRate = 0.7)

pipe = Pipeline(stages = string_indexes + onehot_indexes + [assembler, label_indexes, rf])
```

```
mod = pipe.fit(df)
```

```
df2 = mod.transform(df)
```

The feature importance score that is returned comes in the form of a sparse vector. This is not very human readable and we would need to map this to the actual variable names for some insights. I wrote a little function to return the variable names sorted by importance score as a pandas data frame. This was inspired by the following post on stackoverflow (https://stackoverflow.com/questions/42935914/how-to-map-features-from-the-output-of-a-vectorassembler-back-to-the-column-name)

```
mod.stages[-1].featureImportances
```

```
SparseVector(63, {0: 0.0257, 1: 0.1596, 2: 0.0037, 3: 0.2212, 4: 0.0305, 5: 0.0389, 6: 0.0762, 7: 0.0423, 8: 0.1869,
```

```
def ExtractFeatureImp(featureImp, dataset, featuresCol):
    list_extract = []
    for i in dataset.schema[featuresCol].metadata["ml_attr"]["attrs"]:
        list_extract = list_extract + dataset.schema[featuresCol].metadata["ml_attr"]["attrs"][i]
    varlist = pd.DataFrame(list_extract)
    varlist['score'] = varlist['idx'].apply(lambda x: featureImp[x])
    return(varlist.sort_values('score', ascending = False))
```

```
ExtractFeatureImp(mod.stages[-1].featureImportances, df2, "features").head(10)
```

	idx	name	score
3	3	pdays	0.221203
8	8	euribor3m	0.186892
1	1	duration	0.159579

	idx	name	score
6	6	cons_price_idx	0.076177
9	9	nr_employed	0.063016
7	7	cons_conf_idx	0.042298
5	5	emp_var_rate	0.038875
4	4	previous	0.030470
43	43	OHE_contact_cellular	0.028401
0	0	age	0.025732

Now that we have the most important faatures in a nicely formatted list, we can extract the top 10 features and create a new input vector column with only these variables. Pyspark has a VectorSlicer function that does exactly that. A new model can then be trained just on these 10 variables.

```
varlist = ExtractFeatureImp(mod.stages[-1].featureImportances, df2, "features")
```

```
varidx = [x for x in varlist['idx'][0:10]]
```

```
varidx
```

```
[3, 8, 1, 6, 9, 7, 5, 4, 43, 0]
```

```
slicer = VectorSlicer(inputCol="features", outputCol="features2", indices=varidx)
df3 = slicer.transform(df2)
```

```
df3 = df3.drop('rawPrediction', 'probability', 'prediction')
rf2 = RandomForestClassifier(labelCol="label", featuresCol="features2", seed = 8464,
                             numTrees=10, cacheNodeIds = True, subsamplingRate = 0.7)
mod2 = rf2.fit(df3)
df4 = mod2.transform(df3)
```

Building the estimator function

Now let us learn to build a new pipeline object that makes the above task easy!

First a bit of theory as taken from the ML pipeline documentation (https://spark.apache.org/docs/2.3.0/ml-pipeline.html):

> *DataFrame*: This ML API uses DataFrame from Spark SQL as an ML dataset, which can hold a variety of data types. E.g., a DataFrame could have different columns storing text, feature vectors, true labels, and predictions.
> *Transformer*: A Transformer is an algorithm which can transform one DataFrame into another DataFrame. E.g., an ML model is a Transformer which transforms a DataFrame with features into a DataFrame with predictions.
> *Estimator*: An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model.
> *Pipeline*: A Pipeline chains multiple Transformers and Estimators together to specify an ML workflow.

The important thing to remember is that the pipeline object has two components. The first is the estimator which returns a model and the second is the model/transformer which returns a dataframe.

We begin by coding up the estimator object. The cross-validation function in the previous post (https://www.timlrx.com/2018/04/08/creating-a-custom-cross-validation-function-in-pyspark/) provides a thorough walk-through on creating the estimator object and params needed. In this case, I wanted the function to select either the top n features or based on a certain cut-off so these parameters are included as arguments to the function. An estimator (either decision tree / random forest / gradient boosted trees) is also required as an input.

```
def __init__(self, estimator = None, selectorType = "numTopFeatures",
                 numTopFeatures = 20, threshold = 0.01, outputCol = "features")
```

Given a dataset we can write a fit function that extracts the feature importance scores

```
mod = est.fit(dataset)
dataset2 = mod.transform(dataset)
varlist = ExtractFeatureImp(mod.featureImportances, dataset2, est.getFeaturesCol())
```

Some conditional statements to select the correct indexes that corresponds to the feature we want to extract. This gives us the output of the model - a list of features we want to extract.

```
if (selectorType == "numTopFeatures"):
    varidx = [x for x in varlist['idx'][0:nfeatures]]
elif (selectorType == "threshold"):
    varidx = [x for x in varlist[varlist['score'] > threshold]['idx']]
```

Now for the second part of the problem - we want to take this list of features and create a transform function that returns the dataset with a new column containing our most relevant features. Sounds familiar? This is exactly what the VectorSlicer transformer does. So there is no need to re-invent the wheel and we can just reurn a VectorSlicer with the correct indices to slice.

```
return VectorSlicer(inputCol = est.getFeaturesCol(),
                            outputCol = outputCol,
                            indices = varidx)
```

That concludes our new feature selection estimator! The full code can be obtained here (https://gist.github.com/timlrx/1d5fdb0a43adbbe32a9336ba5c85b1b2).

Putting the new function to the test

Let's try out the new function. I saved it as a file called FeatureImportanceSelector.py. Notice there is a new pipeline object called fis (featureImpSelector). This takes in the first random forest model and uses the feature importance score from it to extract the top 10 variables.

```
from FeatureImportanceSelector import ExtractFeatureImp, FeatureImpSelector
```

```
# one hot encoding and assembling
encoding_var = [i[0] for i in df.dtypes if (i[1]=='string') & (i[0]!='y')]
num_var = [i[0] for i in df.dtypes if ((i[1]=='int') | (i[1]=='double')) & (i[0]!='y')]

string_indexes = [StringIndexer(inputCol = c, outputCol = 'IDX_' + c, handleInvalid = 'keep') for c in encoding_var]
onehot_indexes = [OneHotEncoderEstimator(inputCols = ['IDX_' + c], outputCols = ['OHE_' + c]) for c in encoding_var]
label_indexes = StringIndexer(inputCol = 'y', outputCol = 'label', handleInvalid = 'keep')
assembler = VectorAssembler(inputCols = num_var + ['OHE_' + c for c in encoding_var], outputCol = "features")

rf = RandomForestClassifier(labelCol="label", featuresCol="features", seed = 8464,
                            numTrees=10, cacheNodeIds = True, subsamplingRate = 0.7)
fis = FeatureImpSelector(estimator = rf, selectorType = "numTopFeatures",
                         numTopFeatures = 10, outputCol = "features_subset")
rf2 = RandomForestClassifier(labelCol="label", featuresCol="features_subset", seed = 8464,
                            numTrees=10, cacheNodeIds = True, subsamplingRate = 0.7)

pipe = Pipeline(stages = string_indexes + onehot_indexes + [assembler, label_indexes, fis, rf2])
```

```
pipeline_mod = pipe.fit(df)
```

```
df2 = pipeline_mod.transform(df)
```

```
ExtractFeatureImp(mod.stages[-1].featureImportances, df2, "features_subset")
```

	idx	name	score
3	3	cons_price_idx	0.221203
9	8	OHE_contact_cellular	0.186892
1	1	euribor3m	0.159579
6	6	emp_var_rate	0.076177
8	9	age	0.063016
7	7	previous	0.042298
5	5	cons_conf_idx	0.038875
4	4	nr_employed	0.030470
0	0	pdays	0.025732
2	2	duration	0.003744

10 features as intended and not suprisingly, it matches the top 10 features as generated by our previous non-pipeline method.

Hope you found the tutorial useful and maybe it will inspire you to create more useful extensions for pyspark.

← PREVIOUS POST (HTTPS://WWW.TIMLRX.COM/2018/04/28/STATISTICAL-MUSINGS/)

NEXT POST → (HTTPS://WWW.TIMLRX.COM/2018/08/09/APPLICATIONS-OF-DAGS-IN-CAUSAL-INFERENCE/)

(mailto:timothy.lin@alumni.ubc.ca) (https://github.com/timlrx) (https://linkedin.com/in/timothy-lin-0600ba141)

(https://www.timlrx.com/index.xml)