

databricks Startup_Analysis_Part_Two_Modeling

Startup Analysis

Part Two



```
import pandas as pd
import numpy as np
# Load functionality to manipulate dataframes
from pyspark.sql import functions as fn
import matplotlib.pyplot as plt
from pyspark.sql.functions import stddev, mean, col
# Functionality for computing features
from pyspark.ml import feature, regression, classification, Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import feature, regression, classification, Pipeline
from pyspark.ml.feature import Tokenizer, VectorAssembler, HashingTF,
Word2Vec, StringIndexer, OneHotEncoder
from pyspark.ml import clustering
from itertools import chain
from pyspark.ml.linalg import Vectors, VectorUDT
from pyspark.ml import classification
from pyspark.ml.classification import LogisticRegression,
RandomForestClassifier
from pyspark.ml import evaluation
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark import keyword_only
from pyspark.ml import Transformer
from pyspark.ml.param.shared import HasInputCol, HasOutputCol, Param
```

LOAD AND IMPORT

```
%sh wget https://www.dropbox.com/s/zj133oas3xa1mma/master.csv -nv
```

```
2018-01-17 21:39:15 URL:https://dl.dropboxusercontent.com/content_link/Iq2aU
JBd4t6AvHqQLg8V5USmVzo2LbMDrHXtB10sxNEF1GTNVeTjJ51s9wk0EU30/file [5723844/57
23844] -> "master.csv" [1]
```

```
# load master dataset
dfmaster =
sqlContext.read.format("csv").load("file:///databricks/driver/master.csv",
delimiter = ",", header = True)
```

Preparations and Understanding before Modeling

```
# create a 0/1 column for acquisitions
dfmaster = dfmaster.\
  withColumn("labelacq", fn.when(col("status") ==
"acquired", "1").otherwise("0"))
```

```
# number of rows in master table
print(dfmaster.count())
```

```
49444
```

NAs and market column (with too many levels) handling

```
# check for missing values
dfmaster.toPandas().isnull().sum()
```

```
Out[6]:
permalink          0
name               0
market            3966
status            1314
country_code       5272
city              6115
```

```

funding_rounds          0
founded_year            10955
quarter_new             10955
age                     10955
count_investor          0
time_to_first_funding   24730
total_raised_usd        26342
category_final          0
investor_country_code   20954
funding_round_type      17115
labelacq                0
dtype: int64

```

```

# drop market columns because of too many level and better breakdown with
the category_final column
dfmaster1 = dfmaster.drop("market")

```

```

# drop rows with missing values
dfmaster1drop = dfmaster1.na.drop()

```

We decided to drop the rows with any NULL value, because we still had enough observations and we did not want to value the missing entries equally for every observation. We did one testing and the AUC performance was worse anyways. However, for further analysis it would be worth to try different methods to fill in the missing values.

String indexer, one hot encoder and casting to numerics

```

# create index for categorical variables
# use pipeline to apply indexer
list1 =
["country_code", "city", "quarter_new", "investor_country_code", "funding_round_
type", "category_final"]
indexers = [StringIndexer(inputCol=column,
outputCol=column+"_index").fit(dfmaster1drop) for column in list1]
pipelineindex = Pipeline(stages=indexers).fit(dfmaster1drop)
dfmasternew = pipelineindex.transform(dfmaster1drop)

```

```

# convert string to double for numerical variables
dfmasternew = dfmasternew.\
  withColumn("numeric_funding_rounds",
dfmasternew["funding_rounds"].cast("double")).\
  withColumn("numeric_age", dfmasternew["age"].cast("double")).\
  withColumn("numeric_count_investor",
dfmasternew["count_investor"].cast("double")).\
  withColumn("numeric_time_to_first_funding",
dfmasternew["time_to_first_funding"].cast("double")).\
  withColumn("numeric_total_raised_usd",
dfmasternew["total_raised_usd"].cast("double")).\
  withColumn("label", dfmasternew["labelacq"].cast("double"))

# save
dfone = dfmasternew

#display(dfone)

# list of index columns of categorical variables for the onehotencoder
list2 = dfone.columns[16:22]
list2

Out[13]:
['country_code_index',
 'city_index',
 'quarter_new_index',
 'investor_country_code_index',
 'funding_round_type_index',
 'category_final_index']

# create sparse matrix of indexed categorical columns
# use pipeline to apply the encoder
onehotencoder_stages = [OneHotEncoder(inputCol=c, outputCol='onehotencoded_'
+ c) for c in list2]
pipelineonehot = Pipeline(stages=onehotencoder_stages)
pipeline_mode = pipelineonehot.fit(dfone)
df_coded = pipeline_mode.transform(dfone)

display(df_coded)

```

permalink	name	status	country_code	city	funding_round
/organization/5min	5min Media	acquired	USA	New York	3
/organization/abpathfinder	ABPathfinder	operating	USA	Overland Park	3

/organization/adaptivity	Adaptivity	acquired	USA	Charlotte	6
/organization/aerpio-therapeutics	Aerpio Therapeutics	operating	USA	Cincinnati	4

Showing the first 1000 rows.



Data split, defining vector assemblers & standard scaler and creating labellist

```
# split dataset into training, validation and testing dataset
training_df, validation_df, testing_df = df_coded.randomSplit([0.6, 0.3, 0.1])
```

VECTOR ASSEMBLER

```
training_df.columns[22:27]
```

```
Out[17]:
['numeric_funding_rounds',
 'numeric_age',
 'numeric_count_investor',
 'numeric_time_to_first_funding',
 'numeric_total_raised_usd']
```

```
training_df.columns[28:37]
```

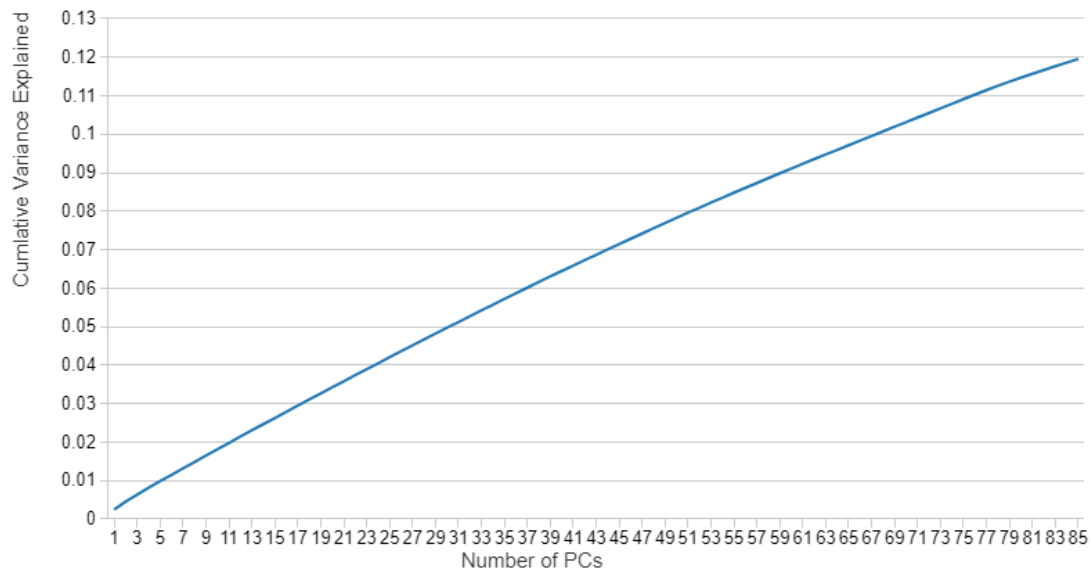
```
Out[18]:
['onehotencoded_country_code_index',
 'onehotencoded_city_index',
 'onehotencoded_quarter_new_index',
 'onehotencoded_investor_country_code_index',
 'onehotencoded_funding_round_type_index',
 'onehotencoded_category_final_index']
```

We chose to only standardize the numeric values, because of interpretability.

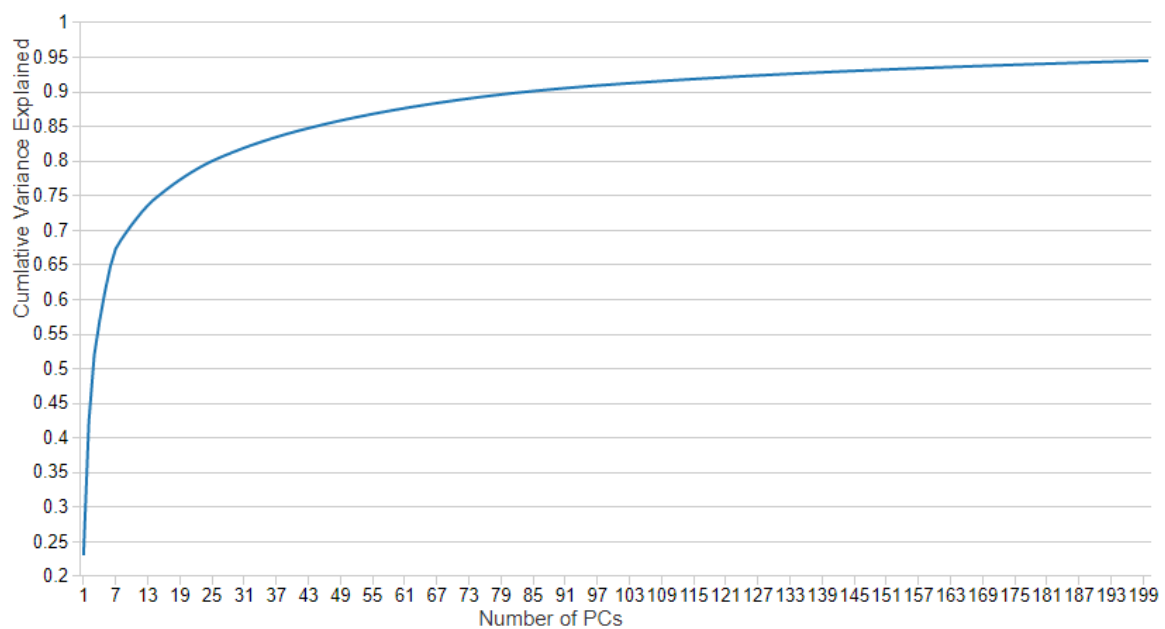
Further, our explained variance when standardizing all of the columns was very strange, because it was very low as seen in the figure below. However, the testing performance was the same if not a little bit better and the weights (for Logistic

Regression) were better in terms of interpretation - you can take a look at it in the additional analysis section at the end of the notebook.

Explained Variance with all columns standardized



Explained Variance with only numerical columns standardized



```
# define vector assembler with the features for the modelling
vanum = VectorAssembler(). \
    setInputCols(training_df.columns[22:27]). \
    setOutputCol('features_nonstd')
```

```

# define vector assembler with the features for the modelling
vacate = VectorAssembler(). \
    setInputCols(training_df.columns[28:37]). \
    setOutputCol('featurescate')

va = VectorAssembler(). \
    setInputCols(['featuresnum', 'featurescate']). \
    setOutputCol('features')

std = feature.StandardScaler(withMean=True,
withStd=True).setInputCol('features_nonstd').setOutputCol('featuresnum')

#display(va.transform(training_df))

# suffix for investor country code because intersection with county_code of
the companies
invcc = ['{}_{}'.format(a, "investor") for a in indexers[3].labels]

# define labellist by using the indexer stages for displaying the weights &
loadings
labellist = training_df.columns[22:27] + indexers[0].labels +
indexers[1].labels + indexers[2].labels + invcc + indexers[4].labels +
indexers[5].labels

training_df.columns[28:37]

Out[26]:
['onehotencoded_country_code_index',
 'onehotencoded_city_index',
 'onehotencoded_quarter_new_index',
 'onehotencoded_investor_country_code_index',
 'onehotencoded_funding_round_type_index',
 'onehotencoded_category_final_index']

```

```

# null dummy for onehotencoded_country_code_index
print("null dummy for onehotencoded_country_code_index")
print(len(indexers[0].labels))
print(indexers[0].labels[79])
# null dummy for onehotencoded_city_index
print("null dummy for onehotencoded_city_index")
print(indexers[1].labels[1761])
print(len(indexers[1].labels))
# null dummy for onehotencoded_quarter_new_index
print("null dummy for onehotencoded_quarter_new_index")
print(len(indexers[2].labels))
print(indexers[2].labels[3])
# null dummy for onehotencoded_investor_country_code_index
print("null dummy for onehotencoded_investor_country_code_index")
print(len(invcc))
print(invcc[67])
# null dummy for onehotencoded_funding_round_type_index
print("null dummy for onehotencoded_funding_round_type_index")
print(len(indexers[4].labels))
print(indexers[4].labels[12])
# null dummy for onehotencoded_category_final_index
print("null dummy for onehotencoded_category_final_index")
print(len(indexers[5].labels))
print(indexers[5].labels[210])

null dummy for onehotencoded_country_code_index
80
SRB
null dummy for onehotencoded_city_index
Aldermaston
1762
null dummy for onehotencoded_quarter_new_index
4
Q4
null dummy for onehotencoded_investor_country_code_index
68
SVK_investor
null dummy for onehotencoded_funding_round_type_index
13
secondary_market
null dummy for onehotencoded_category_final_index
211
Video

```



```
# delete for null dummies from labellist
labellist.remove("SRB")
labellist.remove("Aldermaston")
labellist.remove("Q4")
labellist.remove("SVK_investor")
labellist.remove("secondary_market")
labellist.remove("Video")
```

Modeling

RANDOM FOREST

```
# define binary classification evaluator
bce = BinaryClassificationEvaluator()
```

We chose to run Random Forest models with 20, 15 and 25 trees.

```
# define default, 15 trees and 25 trees random forest classifier
rf = RandomForestClassifier(maxBins=10000, featuresCol='features',
labelCol='label')
rf15 = RandomForestClassifier(numTrees=15, maxBins=10000,
featuresCol='features', labelCol='label')
rf25 = RandomForestClassifier(numTrees=25, maxBins=10000,
featuresCol='features', labelCol='label')

# define and fit pipelines with vector assembler and random forest
classifier
rf_pipeline = Pipeline(stages=[vanum, std, vacate, va, rf]).fit(training_df)
rf_pipeline_15 = Pipeline(stages=[vanum, std, vacate, va,
rf15]).fit(training_df)
rf_pipeline_25 = Pipeline(stages=[vanum, std, vacate, va,
rf25]).fit(training_df)

# apply pipeline to validation dataset
dfrf = rf_pipeline.transform(validation_df)
dfrf_15 = rf_pipeline_15.transform(validation_df)
dfrf_25 = rf_pipeline_25.transform(validation_df)
```

Performance

```
# print the areas under the curve for the different random forest pipelines
print("Random Forest with 20 trees: AUC = {}".format(bce.evaluate(dfrf)))
print("Random Forest 15 trees: AUC = {}".format(bce.evaluate(dfrf_15)))
print("Random Forest 25 trees: AUC = {}".format(bce.evaluate(dfrf_25)))
```

```
Random Forest with 20 trees: AUC = 0.677273562212
```

```
Random Forest 15 trees: AUC = 0.677220220921
```

```
Random Forest 25 trees: AUC = 0.685121750493
```

The performance in terms of AUC is very much the same for all three models. We always also look at the AUC, because our target variable is not really well balanced. AUC will give us the between the TRUE POSITIVE and FALSE POSITIVE rate. The accuracy on the validation dataset seems very good. However, it is strange that the accuracies are exactly the same even though the raw predictions and probabilities are different. The AUC is harder to estimate, because there is no general rule which value is good - 0.71 seems okay though.

```
# print the accuracies for the different random forest pipelines
print(dfrf.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("Accuracy for
Random Forest with 20 trees")).show())
print(dfrf_15.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("Accuracy for
Random Forest with 15 trees")).show())
print(dfrf_25.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("Accuracy for
Random Forest with 25 trees")).show())
```

```
+-----+
|Accuracy for Random Forest with 20 trees|
+-----+
|                                0.8592820964545255|
+-----+
```

None

```
+-----+
|Accuracy for Random Forest with 15 trees|
+-----+
|                                0.8592820964545255|
+-----+
```

None

```
+-----+
|Accuracy for Random Forest with 25 trees|
+-----+
|                                0.8592820964545255|
+-----+
```

None

For some reason the accuracy is exactly the same for all three models, probably meaning that the three models do not have a significant difference. However, the values for the importancies (see down below) are different.

Importancies

```
# create spark df with the 20 highest labels and the corresponding
importancies + sorting by importancy
rfw = spark.createDataFrame(pd.DataFrame(list(zip(labellist,
rf_pipeline.stages[4].featureImportances.toArray()))),
    columns = ['column',
'importancy']).sort_values('importancy').tail(20))
display(rfw)
```

column
Islandia
Neuilly-sur-seine
Seattle
Software
Petaling Jaya
Web Hosting
Mountain View
Jersey City
Advertising



```
# create spark df with the labels and the corresponding importancies +
sorting by importancy
rf15w = spark.createDataFrame(pd.DataFrame(list(zip(labellist,
rf_pipeline_15.stages[4].featureImportances.toArray()))),
    columns = ['column',
'importancy']).sort_values('importancy').tail(20))
display(rf15w)
```

column
Porto Salvo
Advertising

label	weight
Messaging	0
numeric_total_raised_usd	0.01011392343139394
numeric_count_investor	0.07773043377865112
venture	0.21539795559824915
Storage	0.2863519983952415
USA_investor	0.31448037682824714
USA	0.3335017723606627
numeric_age	0.3756817334118509

those were the important features for the best performing RF with the training dataset at that time

The calculation for this figure is in the Logistic Regression part.

The table shows us that a company that is established, is located in the USA, has the majority of investors from the USA and has venture funding and a decent investore count has a good chance to get acquired. Also, the storage market seems to be hot (category)

LOGISTIC REGRESSION

```
# define logistic regression parameters and pipeline stages
lambda_par1 = 0.01
alpha_par1 = 0.05
en_lr1 = LogisticRegression().\
    setLabelCol('label').\
    setFeaturesCol('features').\
    setRegParam(lambda_par1).\
    setElasticNetParam(alpha_par1)

# change the parameters of the second classifier below
lambda_par2 = 0.05
alpha_par2 = 0.05
en_lr2 = LogisticRegression().\
    setLabelCol('label').\
    setFeaturesCol('features').\
    setRegParam(lambda_par2).\
    setElasticNetParam(alpha_par2)

# change the parameters of the thrid classifier below
lambda_par3 = 0.1
alpha_par3 = 0.05
en_lr3 = LogisticRegression().\
    setLabelCol('label').\
    setFeaturesCol('features').\
    setRegParam(lambda_par3).\
    setElasticNetParam(alpha_par3)

en_lr_estimator1 = Pipeline(
    stages=[vanum, std, vacate, va, en_lr1])
en_lr_estimator2 = Pipeline(
    stages=[vanum, std, vacate, va, en_lr2])
en_lr_estimator3 = Pipeline(
    stages=[vanum, std, vacate, va, en_lr3])
```

A function to possibly search for the optimal lambda and alpha

```

#lstauc = []
#lsacc = []
#lsi = []
#lsa = []
#for i in range(20):
#    lambda_par3 = 0.01 + i*0.01
#    for a in range(20):
#        alpha_par3 = 0.01 + i*0.01
#        en_lr3 = LogisticRegression().\
#            setLabelCol('label').\
#            setFeaturesCol('features').\
#            setRegParam(lambda_par3).\
#            setElasticNetParam(alpha_par3)
#        en_lr_estimator3 = Pipeline(stages=[va, en_lr3])
#        en_lr_model3 = en_lr_estimator3.fit(training_df)
#        dfmodel3 = en_lr_model3.transform(validation_df)
#        auc = bce.evaluate(dfmodel3)
#        lstauc.append(auc)
#        acc = dfmodel3.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("acc")).toPan
das().acc[0]
#        lsacc.append(acc)
#        #lsi.append[i]
#        #lsa.append[a]

# fit logistic regression pipelines
en_lr_model1 = en_lr_estimator1.fit(training_df)
en_lr_model2 = en_lr_estimator2.fit(training_df)
en_lr_model3 = en_lr_estimator3.fit(training_df)

# apply pipeline to validation dataset
dfmodel1 = en_lr_model1.transform(validation_df)
dfmodel2 = en_lr_model2.transform(validation_df)
dfmodel3 = en_lr_model3.transform(validation_df)

```

Performance

```

# print areas under the curve of the different logistic regressions
print("Logistic Regression Model 1: AUC =
{}".format(bce.evaluate(dfmodel1)))
print("Logistic Regression Model 2: AUC =
{}".format(bce.evaluate(dfmodel2)))
print("Logistic Regression Model 3: AUC =
{}".format(bce.evaluate(dfmodel3)))

```

```
Logistic Regression Model 1: AUC = 0.735168113299
Logistic Regression Model 2: AUC = 0.73218180316
Logistic Regression Model 3: AUC = 0.719581627816
```

The AUC seems to be better in comparison to Random Forest, while the accuracy is a little bit lower or almost the same.

Logistic regressions seems to perform a little bit better than Random Forest since the AUC is higher.

```
# print accuracies of the different logistic regressions
print(dfmodel1.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("Accuracy for
Model 1"))).show())
print(dfmodel2.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("Accuracy for
Model 2"))).show())
print(dfmodel3.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("Accuracy for
Model 3"))).show())
```

```
+-----+
|Accuracy for Model 1|
+-----+
| 0.8513543272406959|
+-----+
```

None

```
+-----+
|Accuracy for Model 2|
+-----+
| 0.8570799383395727|
+-----+
```

None

```
+-----+
|Accuracy for Model 3|
+-----+
| 0.8595023122660207|
+-----+
```

None

Weights

The weights are almost useless, because the level of categories seems to be too high in some features and therefore, the weights are close to each other and we also have high amount of features with weight 0 (38%). We just use the weights as a reference for the importance of Random Forest, as already described.

```
display(spark.createDataFrame(pd.DataFrame(list(zip(labellist,
en_lr_model1.stages[4].coefficients.toArray()))),
      columns = ['label',
'weight']).sort_values('weight')).agg((fn.count(fn.when(col("weight") == 0,
True)) / fn.count(col("label"))).alias("Ratio"))
```

Ratio
0.40992044922788956



```
# create spark df with the first 10 lowest labels and corresponding weights
pd.DataFrame(list(zip(labellist,
en_lr_model1.stages[4].coefficients.toArray()))),
      columns = ['label', 'weight']).sort_values('weight').head(10)
```

Out[45]:

	label	weight
1328	Irvington	-2.281883
957	Laguna Beach	-2.223327
2029	Electronics	-2.220944
400	Radnor	-2.102383
589	Wallingford	-2.093347
2057	Energy	-2.072233
1828	Great River	-1.984189
355	Marina Del Rey	-1.931526
1891	TWN_investor	-1.926568
942	Merrimack	-1.906096

```
# create spark df with the first 10 highest labels and corresponding weights
pd.DataFrame(list(zip(labellist,
en_lr_model1.stages[4].coefficients.toArray()))),
      columns = ['label', 'weight']).sort_values('weight').tail(10)
```

Out[46]:

	label	weight
1445	Cottesloe	5.446226
1767	Turnhout	5.499689
798	Amherst	5.513969
984	Chestnut Hill	5.533478

1357	Albion Park	5.545395
1055	Sedgefield	5.799024
1521	Wiesbaden	5.865961
1026	Estado De México	6.082296
762	Fuzhou Shi	6.232708
1092	Ivrea	6.484273

```
# create spark df with the first 10 lowest labels and corresponding weights
pd.DataFrame(list(zip(labellist,
en_lr_model2.stages[4].coefficients.toArray()),
columns = ['label', 'weight']).sort_values('weight').head(10)
```

Out[47]:

	label	weight
1952	Consulting	-0.651470
263	Concord	-0.502169
400	Radnor	-0.487080
144	Cleveland	-0.483727
198	Santa Barbara	-0.475748
1996	Big Data Analytics	-0.459632
2011	Startups	-0.459475
191	San Carlos	-0.436617
355	Marina Del Rey	-0.433747
328	Norcross	-0.423684

```
# create spark df with the first 10 highest labels and corresponding weights
pd.DataFrame(list(zip(labellist,
en_lr_model2.stages[4].coefficients.toArray()),
columns = ['label', 'weight']).sort_values('weight').tail(10)
```

Out[48]:

	label	weight
984	Chestnut Hill	2.915277
1249	Bamberg	2.967666
1521	Wiesbaden	2.991406
879	Basel	3.002985
1357	Albion Park	3.031094
1767	Turnhout	3.095172
1055	Sedgefield	3.213030
1092	Ivrea	3.382698
1026	Estado De México	3.413752
762	Fuzhou Shi	3.543553

```
# create spark df with the first lowest labels and corresponding weights
pd.DataFrame(list(zip(labellist,
en_lr_model3.stages[4].coefficients.toArray()),
columns = ['label', 'weight']).sort_values('weight').head(10)
```

Out[49]:

	label	weight
1952	Consulting	-0.326841

```

1928      Biotechnology -0.251262
1942      Education -0.232304
9          CHN -0.226534
1916      seed -0.225028
1938 Clean Technology -0.215883
1947      Manufacturing -0.196722
144      Cleveland -0.170209
1854      CHN_investor -0.169345
12          IND -0.155350

```

```

# create spark df with the first 10 highest labels and corresponding weights
pd.DataFrame(list(zip(labellist,
en_lr_model3.stages[4].coefficients.toArray()),
                columns = ['label', 'weight']).sort_values('weight').tail(10)

```

Out[50]:

	label	weight
380	Redwood Shores	1.688346
1249	Bamberg	1.692812
1521	Wiesbaden	1.698733
1357	Albion Park	1.708755
1445	Cottesloe	1.741397
1767	Turnhout	1.776280
1055	Sedgefield	1.809367
1026	Estado De México	1.903247
1092	Ivrea	1.951242
762	Fuzhou Shi	2.069274

Weights of the important features of the RF with 15 trees as reference

```

# create spark df with all coefficients of LR model 2 (best performance in
terms of AUC and accuracy combined)

```

```

dflrcoeff = spark.createDataFrame(pd.DataFrame(list(zip(labellist,
en_lr_model2.stages[4].coefficients.toArray()),
                columns = ['label', 'weight']).sort_values('weight'))

```

```

# list with the highest 8 importancies based on RF model with 15 trees (best
performance in terms of AUC and accuracy combined)

```

```

rf15imp =
["Storage","Messaging","numeric_count_investor","USA","venture","USA_investo
r","numeric_age","numeric_total_raised_usd"]


```

```

# display weights for the important factors from the RF model with 15 trees
in order to see if its negative or positive
display(dflrcoeff.where(col("label").isin(rf15imp)).orderBy("weight"))

```

label
numeric_total_raised_usd
numeric_count_investor
Messaging
venture
USA_investor
numeric_age
USA
Storage
◀
▶



NEURAL NETWORKS

Standard Model

```
# define neural networks (MultilayerPerceptron) classifier
mlp = classification.MultilayerPerceptronClassifier(seed=0).\
    setStepSize(0.2).\
    setMaxIter(200).\
    setLayers([2137, 2]).\
    setFeaturesCol('features')

# define and fit neural network pipeline
mlp_simple_model = Pipeline(stages=[vanum, std, vacate, va,
mlp]).fit(training_df)

# define evaluators for accuracy and area under the curve
evaluator =
evaluation.MulticlassClassificationEvaluator(metricName="accuracy")
evaluatorauc = evaluation.MulticlassClassificationEvaluator()

# apply pipeline to validation dataset
dfnn = mlp_simple_model.transform(validation_df)

# print accuracy and area under the curve for NN
print(evaluator.evaluate(dfnn))
print(evaluatorauc.evaluate(dfnn))
```

0.843206342215
0.815919476773

Models with Hidden Layers

```
# define neural networks (MultilayerPerceptron) classifier with hidden
layers
mlp2 = classification.MultilayerPerceptronClassifier(seed=0).\
    setStepSize(0.2).\
    setMaxIter(200).\
    setFeaturesCol('features').\
    setLayers([2137,30,30, 2])
mlp3 = classification.MultilayerPerceptronClassifier(seed=0).\
    setStepSize(0.2).\
    setMaxIter(200).\
    setFeaturesCol('features').\
    setLayers([2137,10,10, 2])
mlp4 = classification.MultilayerPerceptronClassifier(seed=0).\
    setStepSize(0.2).\
    setMaxIter(200).\
    setFeaturesCol('features').\
    setLayers([2137,20,20, 2])
mlp5 = classification.MultilayerPerceptronClassifier(seed=0).\
    setStepSize(0.2).\
    setMaxIter(200).\
    setFeaturesCol('features').\
    setLayers([2137,30, 2])
mlp6 = classification.MultilayerPerceptronClassifier(seed=0).\
    setStepSize(0.2).\
    setMaxIter(200).\
    setFeaturesCol('features').\
    setLayers([2137,30,30,30, 2])

# define and fit pipeline
mlp2_model = Pipeline(stages=[vanum, std, vacate, va,
mlp2]).fit(training_df)
mlp3_model = Pipeline(stages=[vanum, std, vacate, va,
mlp3]).fit(training_df)
mlp4_model = Pipeline(stages=[vanum, std, vacate, va,
mlp4]).fit(training_df)
mlp5_model = Pipeline(stages=[vanum, std, vacate, va,
mlp3]).fit(training_df)
mlp6_model = Pipeline(stages=[vanum, std, vacate, va,
mlp4]).fit(training_df)
```

```
# apply and fit model to validation dataset
dfnn2 = mlp2_model.transform(validation_df)
dfnn3 = mlp3_model.transform(validation_df)
dfnn4 = mlp4_model.transform(validation_df)
dfnn5 = mlp3_model.transform(validation_df)
dfnn6 = mlp4_model.transform(validation_df)
```

Performance

The accuracy of the Neural Networks models is a bit worse in comparison to Random Forest and Logistic Regression (on the validation dataset). However, the AUC increases by a lot. This actually might be an indication that the Neural Networks performs the best, regardless of the worse accuracy.

NN Model with 2 hidden layers and 30 neurons each seems to perform the best.

```
# print accuracy and area under the curve
print("NN Model with 2 hidden layers and 30 neurons each: Accuracy =
{}".format(evaluator.evaluate(dfnn2)))
print("NN Model with 2 hidden layers and 30 neurons each: AUC =
{}".format(evaluatorauc.evaluate(dfnn2)))
print("-----
-----")
print("NN Model with 2 hidden layers and 10 neurons each: Accuracy =
{}".format(evaluator.evaluate(dfnn3)))
print("NN Model with 2 hidden layers and 10 neurons each: AUC =
{}".format(evaluatorauc.evaluate(dfnn3)))
print("-----
-----")
print("NN Model with 2 hidden layers and 20 neurons each: Accuracy =
{}".format(evaluator.evaluate(dfnn4)))
print("NN Model with 2 hidden layers and 20 neurons each: AUC =
{}".format(evaluatorauc.evaluate(dfnn4)))
print("-----
-----")
print("NN Model with 1 hidden layer and 30 neurons: Accuracy =
{}".format(evaluator.evaluate(dfnn5)))
print("NN Model with 1 hidden layer and 30 neurons: AUC =
{}".format(evaluatorauc.evaluate(dfnn5)))
print("-----
-----")
print("NN Model with 3 hidden layers and 30 neurons each: Accuracy =
{}".format(evaluator.evaluate(dfnn6)))
print("NN Model with 3 hidden layers and 30 neurons each: AUC =
{}".format(evaluatorauc.evaluate(dfnn6)))
```

```

NN Model with 2 hidden layers and 30 neurons each: Accuracy = 0.819863466197
NN Model with 2 hidden layers and 30 neurons each: AUC = 0.81144334681
-----
NN Model with 2 hidden layers and 10 neurons each: Accuracy = 0.817220876459
NN Model with 2 hidden layers and 10 neurons each: AUC = 0.81017995708
-----
NN Model with 2 hidden layers and 20 neurons each: Accuracy = 0.813256991852
NN Model with 2 hidden layers and 20 neurons each: AUC = 0.815835240669
-----
NN Model with 1 hidden layer and 30 neurons: Accuracy = 0.817220876459
NN Model with 1 hidden layer and 30 neurons: AUC = 0.81017995708
-----
NN Model with 3 hidden layers and 30 neurons each: Accuracy = 0.813256991852
NN Model with 3 hidden layers and 30 neurons each: AUC = 0.815835240669

```

PCA

Decision of the right Number of PCs

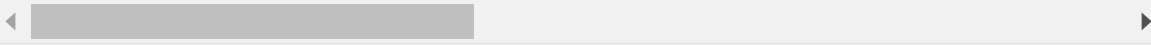

We also performed a dimensional reduction due to the high number of feature, which is mainly caused by the levels of categorical features. We chose to use 200 PCs as a base and then to look at the cumulative variance. We found out that roughly 85 PCs can explain 90% of the variance.

```

# define and fit a PCA model wit k = 200 as reference
#
pcavar = feature.PCA(k=200, inputCol='features', outputCol='pca_feat')
pca_var = Pipeline(
    stages=[vanum, std, vacate, va, pcavar])
pca_var_model = pca_var.fit(df_coded)
varlist = pca_var_model.stages[4].explainedVariance
npvar = np.cumsum(varlist)
pci = [pci for pci in range(1,201,1)]
dfvar = spark.createDataFrame(pd.DataFrame({"Number of PCs": pci,
"Cumulative Variance Explained": npvar}))
display(dfvar)

```

Cumulative Variance Explained
0.23123319056868516
0.4281109031063016
0.520467571554613
0.5689963956588386

0.610880525713373
0.6468855367181202
0.6726937077608065



So we reach a explained variance of 90% with roughly 85 PCs

Modeling

```
# define pca feature function with k = 85
pca = feature.PCA(k=85, inputCol='features', outputCol='pca_feat')
```



```
# define the parameters and pipelines for the logistic regressions with PCA
features
lambda_par1 = 0.01
alpha_par1 = 0.05
en_lr1_pca = LogisticRegression().\
    setLabelCol('label').\
    setFeaturesCol('pca_feat').\
    setRegParam(lambda_par1).\
    setElasticNetParam(alpha_par1)

# change the parameters of the second classifier below
lambda_par2 = 0.05
alpha_par2 = 0.05
en_lr2_pca = LogisticRegression().\
    setLabelCol('label').\
    setFeaturesCol('pca_feat').\
    setRegParam(lambda_par2).\
    setElasticNetParam(alpha_par2)

# change the parameters of the thrid classifier below
lambda_par3 = 0.1
alpha_par3 = 0.05
en_lr3_pca = LogisticRegression().\
    setLabelCol('label').\
    setFeaturesCol('pca_feat').\
    setRegParam(lambda_par3).\
    setElasticNetParam(alpha_par3)

en_lr_estimator1_pca = Pipeline(
    stages=[vanum, std, vacate, va, pca, en_lr1])
en_lr_estimator2_pca = Pipeline(
    stages=[vanum, std, vacate, va, pca, en_lr2])
en_lr_estimator3_pca = Pipeline(
    stages=[vanum, std, vacate, va, pca, en_lr3])

# fit the PCA logistic regression pipelines
en_lr_model1_pca = en_lr_estimator1_pca.fit(training_df)
en_lr_model2_pca = en_lr_estimator2_pca.fit(training_df)
en_lr_model3_pca = en_lr_estimator3_pca.fit(training_df)

# apply the pipelines to the validation dataset
dfmodel1_pca = en_lr_model1_pca.transform(validation_df)
dfmodel2_pca = en_lr_model2_pca.transform(validation_df)
dfmodel3_pca = en_lr_model3_pca.transform(validation_df)

#display(dfmodel2_pca)
```

Performance

For some, the numbers of the AUC and accuracies are the same as of the Logistic Regression. The same applies to the raw predictions and probabilities. This might mean that the dimensional reduction has basically no impact at all.

So the performance of the Logistic Regression with PCA has the same conclusion as the normla Logistic Regression.

We feel like there is a mistake, but we can not find it

```
display(dfmodel2_pca)
```

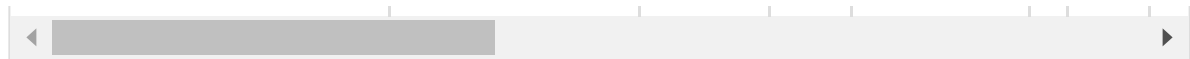
permalink	name	status	country	city	code	funding	year	quarter
/organization/1000museums-com	1000museums.com	operating	USA	Bellevue		6	2008	Q1
/organization/121cast	121cast	operating	AUS	Melbourne		2	2012	Q1
/organization/121nexus	121nexus	operating	USA	Providence		6	2011	Q4
/organization/140fire	140Fire	operating	USA	Santa Monica		1	2010	Q1

Showing the first 399 rows.



```
display(dfmodel2)
```

permalink	name	status	country	city	code	funding	year	quarter
/organization/1000museums-com	1000museums.com	operating	USA	Bellevue		6	2008	Q1
/organization/121cast	121cast	operating	AUS	Melbourne		2	2012	Q1
/organization/121nexus	121nexus	operating	USA	Providence		6	2011	Q4



Showing the first 1000 rows.



```
# print the areas under the curve and accuracies of the different PCA
logistic regression models
print("Logistic Regression Model 1: AUC =
{}".format(bce.evaluate(dfmodel1_pca)))
print("Logistic Regression Model 2: AUC =
{}".format(bce.evaluate(dfmodel2_pca)))
print("Logistic Regression Model 3: AUC =
{}".format(bce.evaluate(dfmodel3_pca)))
#
print(dfmodel1_pca.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("Accuracy for
Model 1"))).show())
print(dfmodel2_pca.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("Accuracy for
Model 2"))).show())
print(dfmodel3_pca.select(fn.expr('float(label =
prediction)').alias('correct')).select(fn.avg('correct').alias("Accuracy for
Model 3"))).show())
```

Logistic Regression Model 1: AUC = 0.735168113299

Logistic Regression Model 2: AUC = 0.73218180316

Logistic Regression Model 3: AUC = 0.719581627816

```
+-----+
|Accuracy for Model 1|
+-----+
| 0.8513543272406959|
+-----+
```

None

```
+-----+
|Accuracy for Model 2|
+-----+
| 0.8570799383395727|
+-----+
```

None

```
+-----+
|Accuracy for Model 3|
+-----+
| 0.8595023122660207|
+-----+
```

None

Exploring Loadings