svm_model

July 1, 2021

1 SVM Classification

Isaac Stevens is3sb

[80]:
```
from pyspark.sql import SparkSession
from pyspark.sql.types import ArrayType, StructField, StructType, StringType,␣
 ↪IntegerType
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import Correlation
from pyspark.sql import functions as F
from pyspark.sql.functions import col
from pyspark.sql.types import *
from pyspark.sql import SQLContext
from pyspark.ml.tuning import CrossValidator,␣
 ↪ParamGridBuilder,TrainValidationSplit
from pyspark.ml.evaluation import␣
 ↪BinaryClassificationEvaluator,MulticlassClassificationEvaluator
from pyspark.ml.classification import LinearSVC
from pyspark.ml import Pipeline
from pyspark.ml.feature import PCA
from pyspark.mllib.evaluation import␣
 ↪MulticlassMetrics,BinaryClassificationMetrics
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
import pandas as pd
pd.set_option('display.max_rows', 200000)
```

[2]:
```
# set up the session
spark = SparkSession \
        .builder \
        .appName("project")\
        .config("spark.executor.memory", "100g")\
        .getOrCreate()

sqlContext = SQLContext(spark)
```

[6]:
```
%%time
#import whole data from the census
data = spark.read.csv('/project/ds5559/ds5110_project_snoo/acs_15_19_south.
 ↪csv', inferSchema="true", header="true")
```

```
CPU times: user 4.45 ms, sys: 1.76 ms, total: 6.21 ms
Wall time: 31.9 s
```

[7]:
```
#udf for education flag
def EDUCFunc(value):
  if   value > 6:
      return 1
  else:
      return 0
```

[8]:
```
#create the function to be applied and create a new column EDUC_FLAG
udfsomefunc = F.udf(EDUCFunc, IntegerType())
data = data.withColumn("EDUC_FLAG", udfsomefunc("EDUC"))
#see sample data
data.select('EDUC_FLAG').show(5)
```

```
+---------+
|EDUC_FLAG|
+---------+
|        0|
|        1|
|        0|
|        1|
|        0|
+---------+
only showing top 5 rows
```

[9]:
```
df = data.withColumn("label",data.EDUC_FLAG).drop("EDUC_FLAG")
```

[76]:
```
#saving col names in case if we can use it later ot iterate or use the list for␣
 ↪labels etc.
cols = df.columns
```

1.1 EDA

[10]:
```
#displaying number of rows and columns in the data
print((df.count(), len(df.columns)))
```

```
(5965249, 206)
```

[11]:
```
#number of years in the data set
df.select('MULTYEAR').distinct().show()
```

```
+--------+
|MULTYEAR|
+--------+
|    2018|
|    2015|
|    2019|
|    2016|
|    2017|
+--------+
```

[12]:
```
#Sample data with seed 42
sampled = df.sampleBy("MULTYEAR", fractions={2015: 0.1, 2016: 0.1, 2017:0.1,␣
 ↪2018:0.1, 2019:0.1}, seed=42)
sampled.groupBy("MULTYEAR").count().orderBy("MULTYEAR").show()
```

```
+--------+------+
|MULTYEAR| count|
+--------+------+
|    2015|117141|
|    2016|117882|
|    2017|119767|
|    2018|119761|
|    2019|121997|
+--------+------+
```

[45]:
```
hhtype_groups = sampled.groupBy("HHTYPE").count().sort(col("count").desc())
```

[51]:
```
#udf to map hhtype
def mapHhtype(value):
    hhtype_dict = {0:'N/A',\
            1: 'Married-couple family household',\
            2: 'Male householder, no wife present',\
            3: 'Female householder, no husband present',\
            4: 'Male householder, living alone',\
            5: 'Male householder, not living alone',\
            6: 'Female householder, living alone',\
            7: 'Female householder, not living alone',\
            9: 'HHTYPE could not be determined'}
    return hhtype_dict.get(value)
```

[62]:
```
hhtype_function = F.udf(mapHhtype, StringType())
```

```
hhtype_groups = hhtype_groups.withColumn("hhtype_long",␣
 ↪hhtype_function("hhtype"))
hhtype_df = hhtype_groups.toPandas()
```

[63]:
```
hhtype_df
```

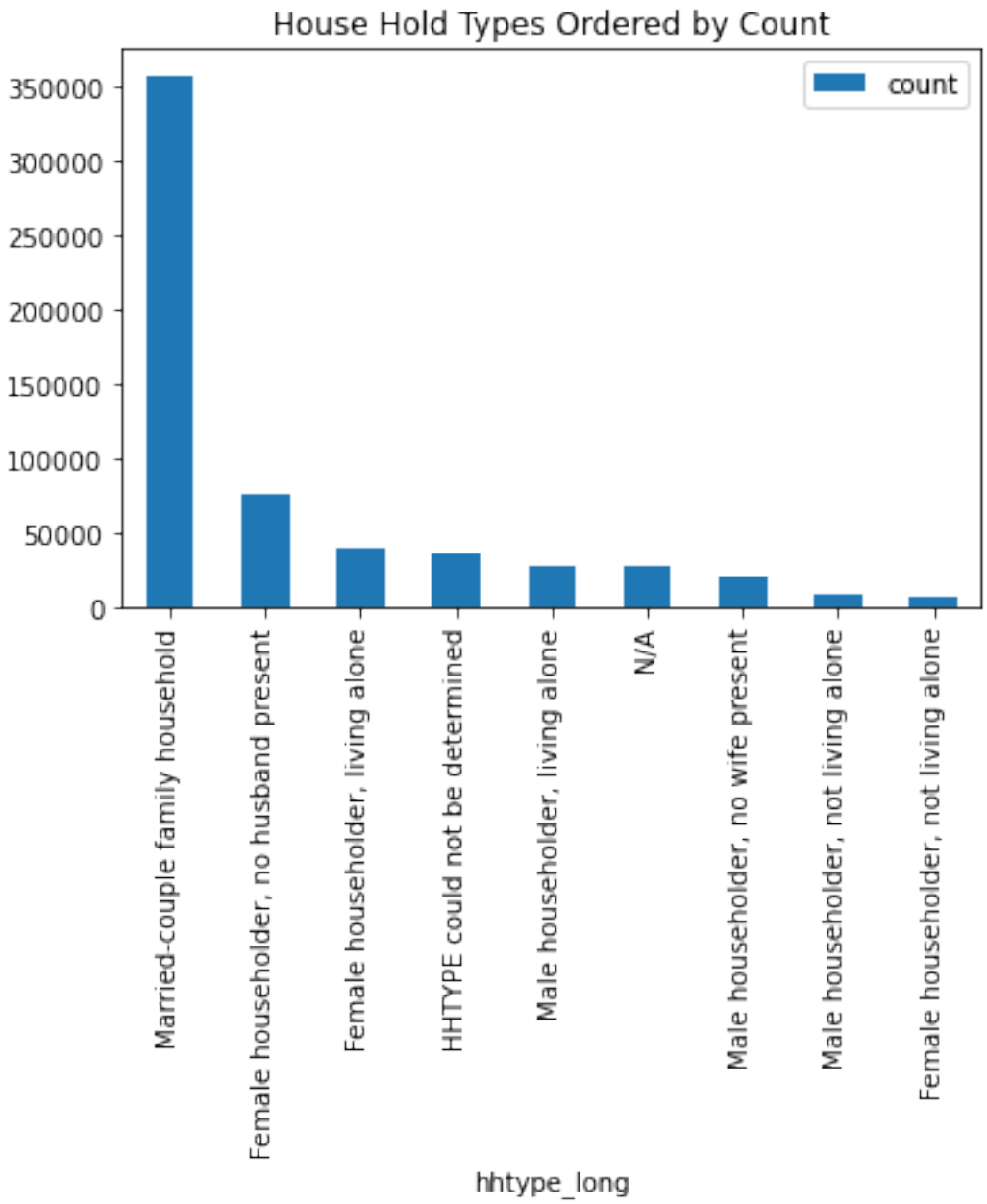
[63]:

	HHTYPE	count	hhtype_long
0	1	357152	Married-couple family household
1	3	75270	Female householder, no husband present
2	6	38840	Female householder, living alone
3	9	35204	HHTYPE could not be determined
4	4	27684	Male householder, living alone
5	0	27627	N/A
6	2	20921	Male householder, no wife present
7	5	7596	Male householder, not living alone
8	7	6254	Female householder, not living alone

[65]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

[74]:
```
hhtype_df.plot.bar(x='hhtype_long', y='count', title = "House Hold Types␣
 ↪Ordered by Count")
```

[74]:
```
<AxesSubplot:title={'center':'House Hold Types Ordered by Count'},
xlabel='hhtype_long'>
```



1.2 Transform Data; Scale; PCA; SVM Classification - seed 42

[77]:
```
%%time
#pass all the features into vector assembler to create a vector format to pass␣
 ↪tto the classification model
assembler = VectorAssembler(inputCols=[cols for cols in cols if cols!='label'],␣
 ↪outputCol="features")
transformed = assembler.transform(sampled)
```

```
#register table as sql table and keep only columns fo interest and save in a␣
 ↪new dataframe. This can be done without using SQl as well.
transformed.registerTempTable('transformed_tbl')
transformed_df = sqlContext.sql('select label,features from transformed_tbl')
transformed_df.show(5)
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|    1|(205,[0,1,2,3,4,5…|
|    0|(205,[0,1,2,3,4,5…|
|    1|(205,[0,1,2,3,4,5…|
|    0|(205,[0,1,2,3,4,5…|
|    0|(205,[0,1,2,3,4,5…|
+-----+--------------------+
only showing top 5 rows

CPU times: user 10.5 ms, sys: 2.11 ms, total: 12.6 ms
Wall time: 4.55 s
```

[78]:
```
%%time
#train test split
training_data, test_data = transformed_df.randomSplit([0.7, 0.3], seed=42)
cached_tr = training_data.cache()
```

```
CPU times: user 669 µs, sys: 1.38 ms, total: 2.05 ms
Wall time: 99.6 ms
```

[79]:
```
%%time
#scale the data
scaler_train = StandardScaler(inputCol="features", outputCol="scaledFeatures")
scalerModel_train = scaler_train.fit(cached_tr)
scaledData_train = scalerModel_train.transform(cached_tr)
```

```
CPU times: user 7.59 ms, sys: 2.43 ms, total: 10 ms
Wall time: 29.3 s
```

[83]:
```
%%time
#pca to reduce 200 odd features into principal components - on training data␣
 ↪only because that is our model
#this takes a while to run. imagine it is running at least 9 combinations␣
 ↪models with 3 folds and picking the best. Reduce parameters or folds if you␣
 ↪want it to run faster
pca_model = PCA(inputCol = "scaledFeatures", outputCol = "pca_features_cv")

#create a SVM classifier model to pass into pipeline
```

```
lsvc = LinearSVC(labelCol = "label", featuresCol = "pca_features_cv",␣
 ↪maxIter=10, regParam=0.1)

#creating a pipeline with the pca and model to use in the cross validator
ppl_cv = Pipeline(stages = [pca_model, lsvc])
```

```
CPU times: user 782 µs, sys: 1.44 ms, total: 2.22 ms
Wall time: 4.61 ms
```

[84]:
```
#create a param grid to pass to cross validator
#k --> number of principal components
#number of treess in rf
#need to add more later
paramGrid = ParamGridBuilder() \
  .addGrid(pca_model.k, [10, 20, 30]) \
  .addGrid(lsvc.regParam, [0.1, 0.01]) \
  .build()

#passs the model with variosu combinations of the parameters and it will pick␣
 ↪the best one. Using 3 folds to save time. Check seed=42.
crossval = CrossValidator(estimator = ppl_cv,\
                                      estimatorParamMaps=paramGrid,\
                                      evaluator =␣
 ↪MulticlassClassificationEvaluator(),\
                                      numFolds= 3,seed=42)


#this is our best model - fit the training data
cv_model = crossval.fit(scaledData_train)
```

[85]:
```
#all the 9 model accuracies. The max one was picked as best
avgMetricsGrid = cv_model.avgMetrics
print(avgMetricsGrid)

#https://tsmatz.github.io/azure-databricks-exercise/
 ↪exercise04-hyperparams-tuning.html
#https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.
 ↪tuning.CrossValidator.html
# View all results (accuracy) by each params - these can be converted to pretty␣
 ↪tables in pandas later
list(zip(cv_model.getEstimatorParamMaps()))
```

```
[0.43399249197817313, 0.43399249197817313, 0.43399249197817313,
0.43399249197817313, 0.43399249197817313, 0.43399249197817313]
```

[85]:
```
[({Param(parent='PCA_270b1c8b4933', name='k', doc='the number of principal
components'): 10,
```

```
   Param(parent='LinearSVC_3eaeb6386879', name='regParam', doc='regularization
parameter (>= 0).'): 0.1},),
 ({Param(parent='PCA_270b1c8b4933', name='k', doc='the number of principal
components'): 10,
   Param(parent='LinearSVC_3eaeb6386879', name='regParam', doc='regularization
parameter (>= 0).'): 0.01},),
 ({Param(parent='PCA_270b1c8b4933', name='k', doc='the number of principal
components'): 20,
   Param(parent='LinearSVC_3eaeb6386879', name='regParam', doc='regularization
parameter (>= 0).'): 0.1},),
 ({Param(parent='PCA_270b1c8b4933', name='k', doc='the number of principal
components'): 20,
   Param(parent='LinearSVC_3eaeb6386879', name='regParam', doc='regularization
parameter (>= 0).'): 0.01},),
 ({Param(parent='PCA_270b1c8b4933', name='k', doc='the number of principal
components'): 30,
   Param(parent='LinearSVC_3eaeb6386879', name='regParam', doc='regularization
parameter (>= 0).'): 0.1},),
 ({Param(parent='PCA_270b1c8b4933', name='k', doc='the number of principal
components'): 30,
   Param(parent='LinearSVC_3eaeb6386879', name='regParam', doc='regularization
parameter (>= 0).'): 0.01},)]
```

[86]:
```
#scale test data
scaler_test = StandardScaler(inputCol="features", outputCol="scaledFeatures")
scalerModel_test = scaler_test.fit(test_data)
scaledData_test = scalerModel_test.transform(test_data)
```

[ ]:
```
%%time
#predict and evaluate the model for accuracy
predictions = cv_model.transform(scaledData_test)
evaluator= MulticlassClassificationEvaluator(labelCol = "label", metricName=␣
 ↪"accuracy")
accuracy = evaluator.evaluate(predictions)
```

[ ]:
```
#SVM accuracy
print(accuracy)
```

[ ]:
```

```