Data Science with PySpark

David Kearney

Aug 31, 2020

CONTENTS

Data Science with PySpark, written by David R. Kearney.

Note: Data Science with PySpark includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

The data used by this book was developed by [**?**].

CHAPTER ONE

RESILIENT DISTRIBUTED DATASETS

Spark uses Java Virtual Machine (JVM) objects Resilient Distributed Datasets (RDD) which are calculated and stored in memory.

1.1 Pyspark Regression with Fiscal Data

> "A minimal example of using Pyspark for Linear Regression"

- toc: true- branch: master- badges: true
- comments: true
- author: David Kearney
- categories: [pyspark, jupyter]
- description: A minimal example of using Pyspark for Linear Regression
- title: Pyspark Regression with Fiscal Data

1.1.1 Bring in needed imports

```
from pyspark.sql.functions import col
from pyspark.sql.types import StringType,BooleanType,DateType,IntegerType
from pyspark.sql.functions import *
```

1.1.2 Load data from CSV

```
#collapse-hide

# Load data from a CSV
file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").option("inferSchema", True).option("header", True).
↪load(file_location)
display(df.take(5))
```

```
df.createOrReplaceTempView("fiscal_stats")

sums = spark.sql("""
select year, sum(it) as total_yearly_it, sum(fr) as total_yearly_fr
from fiscal_stats
```

(continues on next page)

(continued from previous page)

```
group by 1
order by year asc
""")

sums.show()
```

1.1.3 Describing the Data

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

1.1.4 Cast Data Type

```
df2 = df.withColumn("gdp",col("gdp").cast(IntegerType())) \
.withColumn("specific",col("specific").cast(IntegerType())) \
.withColumn("general",col("general").cast(IntegerType())) \
.withColumn("year",col("year").cast(IntegerType())) \
.withColumn("fdi",col("fdi").cast(IntegerType())) \
.withColumn("rnr",col("rnr").cast(IntegerType())) \
.withColumn("rr",col("rr").cast(IntegerType())) \
.withColumn("i",col("i").cast(IntegerType())) \
.withColumn("fr",col("fr").cast(IntegerType()))
```

1.1.5 printSchema

```
df2.printSchema()
```

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression

assembler = VectorAssembler(inputCols=['gdp', 'fdi'], outputCol="features")
train_df = assembler.transform(df2)
```

```
train_df.select("specific", "year").show()
```

1.1.6 Linear Regression in Pyspark

```
lr = LinearRegression(featuresCol = 'features', labelCol='it')
lr_model = lr.fit(train_df)

trainingSummary = lr_model.summary
print("Coefficients: " + str(lr_model.coefficients))
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("R2: %f" % trainingSummary.r2)
```

```
lr_predictions = lr_model.transform(train_df)
lr_predictions.select("prediction","it","features").show(5)
from pyspark.ml.evaluation import RegressionEvaluator
```

(continues on next page)

(continued from previous page)

```
lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
                 labelCol="it",metricName="r2")
```

```
print("R Squared (R2) on test data = %g" % lr_evaluator.evaluate(lr_predictions))
```

```
print("numIterations: %d" % trainingSummary.totalIterations)
print("objectiveHistory: %s" % str(trainingSummary.objectiveHistory))
trainingSummary.residuals.show()
```

```
predictions = lr_model.transform(test_df)
predictions.select("prediction","it","features").show()
```

```
from pyspark.ml.regression import DecisionTreeRegressor
dt = DecisionTreeRegressor(featuresCol ='features', labelCol = 'it')
dt_model = dt.fit(train_df)
dt_predictions = dt_model.transform(train_df)
dt_evaluator = RegressionEvaluator(
    labelCol="it", predictionCol="prediction", metricName="rmse")
rmse = dt_evaluator.evaluate(dt_predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
from pyspark.ml.regression import GBTRegressor
gbt = GBTRegressor(featuresCol = 'features', labelCol = 'it', maxIter=10)
gbt_model = gbt.fit(train_df)
gbt_predictions = gbt_model.transform(train_df)
gbt_predictions.select('prediction', 'it', 'features').show(5)


gbt_evaluator = RegressionEvaluator(
    labelCol="it", predictionCol="prediction", metricName="rmse")
rmse = gbt_evaluator.evaluate(gbt_predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

This post includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

1.2 Group By and Aggregation with Pyspark

"Group By and Aggregation with Pyspark"

- toc: true- branch: master- badges: true
- comments: true
- author: David Kearney
- categories: [pyspark, jupyter]
- description: Group By and Aggregation with Pyspark
- title: Group By and Aggregation with Pyspark

1.2.1 Read CSV and inferSchema

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import countDistinct, avg,stddev


# Load data from a CSV
file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").option("inferSchema", True).option("header", True).
↪load(file_location)
display(df.take(5))
```

```
df.printSchema()
```

1.2.2 Using groupBy for Averages and Counts

```
df.groupBy("province")
```

```
df.groupBy("province").mean().show()
```

```
df.groupBy("reg").mean().show()
```

```
# Count
df.groupBy("reg").count().show()
```

```
# Max
df.groupBy("reg").max().show()
```

```
# Min
df.groupBy("reg").min().show()
```

```
# Sum
df.groupBy("reg").sum().show()
```

```
# Max it across everything
df.agg({'specific':'max'}).show()
```

```
grouped = df.groupBy("reg")
grouped.agg({"it":'max'}).show()
```

```
df.select(countDistinct("reg")).show()
```

```
df.select(countDistinct("reg").alias("Distinct Region")).show()
```

```
df.select(avg('specific')).show()
```

```
df.select(stddev("specific")).show()
```

1.2.3 Choosing Significant Digits with format_number

```
from pyspark.sql.functions import format_number
```

```
specific_std = df.select(stddev("specific").alias('std'))
specific_std.show()
```

```
specific_std.select(format_number('std',0)).show()
```

1.2.4 Using orderBy

```
df.orderBy("specific").show()
```

```
df.orderBy(df["specific"].desc()).show()
```

This post includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

1.3 Handling Missing Data with Pyspark

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import countDistinct, avg,stddev

# Load data from a CSV
file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").option("inferSchema", True).option("header", True).
↪load(file_location)
display(df.take(5))
```

```
df.show()
```

1.3.1 Dropping Columns without non-null values

```
# Has to have at least 2 NON-null values
df.na.drop(thresh=2).show()
```

1.3.2 Dropping any row that contains missing data

```
df.na.drop().show()
```

```
df.na.drop(subset=["general"]).show()
```

```
df.na.drop(how='any').show()
```

```
df.na.drop(how='all').show()
```

1.3.3 Imputation of Null Values

```
df.na.fill('example').show()
```

Imputation of 0

```
df.na.fill(0).show()
```

```
df.na.fill('example',subset=['fr']).show()
```

```
df.na.fill(0,subset=['general']).show()
```

Imputation of the Mean

```
from pyspark.sql.functions import mean
mean_val = df.select(mean(df['general'])).collect()
```

```
mean_val[0][0]
```

```
mean_gen = mean_val[0][0]
```

```
df.na.fill(mean_gen,["general"]).show()
```

```
df.na.fill(df.select(mean(df['general'])).collect()[0][0],['general']).show()
```

This post includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

1.4 Dataframe Filitering and Operations with Pyspark

```
from pyspark.sql import SparkSession

# Load data from a CSV
file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").option("inferSchema", True).option("header", True).
↪load(file_location)
display(df.take(5))
```

1.4.1 Filtering on values in a column

```
df.filter("specific<10000").show()
```

```
df.filter("specific<10000").select('province').show()
```

```
df.filter("specific<10000").select(['province','year']).show()
df.filter(df["specific"] < 10000).show()
```

1.4.2 Filtering on values in 2+ columns

```
df.filter((df["specific"] < 55000) & (df['gdp'] > 200) ).show()
```

```
df.filter((df["specific"] < 55000) | (df['gdp'] > 20000) ).show()
```

```
df.filter((df["specific"] < 55000) & ~(df['gdp'] > 20000) ).show()
```

```
df.filter(df["specific"] == 8964.0).show()
```

```
df.filter(df["province"] == "Zhejiang").show()
```

```
df.filter(df["specific"] == 8964.0).collect()
```

```
result = df.filter(df["specific"] == 8964.0).collect()
```

```
type(result[0])
```

```
row = result[0]
```

```
row.asDict()
```

```
for item in result[0]:
    print(item)
```

This post includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

1.5 Dataframes, Formatting, Casting Data Type and Correlation with Pyspark

```
from pyspark.sql import SparkSession

# Load data from a CSV
file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").option("inferSchema", True).option("header", True).
↪load(file_location)
display(df.take(5))
```

```
df.columns
```

```
df.printSchema()
```

```
# for row in df.head(5):
#     print(row)
#     print('\n')
```

```
df.describe().show()
```

```
df.describe().printSchema()
```

1.5.1 Casting Data Types and Formatting Significant Digits

```
from pyspark.sql.functions import format_number
```

```
result = df.describe()
result.select(result['province']
,format_number(result['specific'].cast('float'),2).alias('specific')
,format_number(result['general'].cast('float'),2).alias('general')
,format_number(result['year'].cast('int'),2).alias('year'),format_number(result['gdp
↪'].cast('float'),2).alias('gdp')
,format_number(result['rnr'].cast('int'),2).alias('rnr'),format_number(result['rr'].
↪cast('float'),2).alias('rr')
,format_number(result['fdi'].cast('int'),2).alias('fdi'),format_number(result['it'].
↪cast('float'),2).alias('it')
,result['reg'].cast('string').alias('reg')
             ).show()
```

1.5.2 New Columns generated from extant columns using withColumn

```
df2 = df.withColumn("specific_gdp_ratio",df["specific"]/(df["gdp"]*100))#.show()
```

```
df2.select('specific_gdp_ratio').show()
```

```
df.orderBy(df["specific"].asc()).head(1)[0][0]
```

1.5.3 Finding the Mean, Max, and Min

```
from pyspark.sql.functions import mean
df.select(mean("specific")).show()
```

```
from pyspark.sql.functions import max,min
```

```
df.select(max("specific"),min("specific")).show()
```

```
df.filter("specific < 60000").count()
```

```
df.filter(df['specific'] < 60000).count()
```

```
from pyspark.sql.functions import count
result = df.filter(df['specific'] < 60000)
result.select(count('specific')).show()
```

```
(df.filter(df["gdp"]>8000).count()*1.0/df.count())*100
```

```
from pyspark.sql.functions import corr
df.select(corr("gdp","fdi")).show()
```

1.5.4 Finding the max value by Year

```
from pyspark.sql.functions import year
#yeardf = df.withColumn("Year",year(df["year"]))
```

```
max_df = df.groupBy('year').max()
```

```
max_df.select('year','max(gdp)').show()
```

```
from pyspark.sql.functions import month
```

```
#df.select("year","avg(gdp)").orderBy('year').show()
```

This post includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

1.6 RDDs and Schemas and Data Types with Pyspark

```
from pyspark.sql import SparkSession

# Load data from a CSV
file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").option("inferSchema", True).option("header", True).
↪load(file_location)
display(df.take(5))
```

```
df.show()
```

```
df.printSchema()
```

```
df.columns
```

```
df.describe()
```

1.6.1 Setting Data Schema and Data Types

```
from pyspark.sql.types import StructField,StringType,IntegerType,StructType
```

```
data_schema = [
StructField("_c0", IntegerType(), True)
,StructField("province", StringType(), True)
,StructField("specific", IntegerType(), True)
,StructField("general", IntegerType(), True)
,StructField("year", IntegerType(), True)
,StructField("gdp", IntegerType(), True)
,StructField("fdi", IntegerType(), True)
,StructField("rnr", IntegerType(), True)
,StructField("rr", IntegerType(), True)
,StructField("i", IntegerType(), True)
,StructField("fr", IntegerType(), True)
,StructField("reg", StringType(), True)
,StructField("it", IntegerType(), True)
]
```

```
final_struc = StructType(fields=data_schema)
```

1.6.2 Applying the Data Schema/Data Types while reading in a CSV

```
df = spark.read.format("CSV").schema(final_struc).load(file_location)
```

```
df.printSchema()
```

```
df.show()
```

```
df['fr']
```

```
type(df['fr'])
```

```
df.select('fr')
```

```
type(df.select('fr'))
```

```
df.select('fr').show()
```

```
df.head(2)
```

```
df.select(['reg','fr'])
```

1.6.3 Using select with RDDs

```
df.select(['reg','fr']).show()
```

```
df.withColumn('fiscal_revenue',df['fr']).show()
```

```
df.show()
```

1.6.4 Renaming Columns using withColumnRenamed

```
df.withColumnRenamed('fr','new_fiscal_revenue').show()
```

1.6.5 New Columns by Transforming extant Columns using withColumn

```
df.withColumn('double_fiscal_revenue',df['fr']*2).show()
```

```
df.withColumn('add_fiscal_revenue',df['fr']+1).show()
```

```
df.withColumn('half_fiscal_revenue',df['fr']/2).show()
```

```
df.withColumn('half_fr',df['fr']/2)
```

1.6.6 Spark SQL for SQL functionality using createOrReplaceTempView

```
df.createOrReplaceTempView("economic_data")
```

```
sql_results = spark.sql("SELECT * FROM economic_data")
```

```
sql_results
```

```
sql_results.show()
```

```
spark.sql("SELECT * FROM economic_data WHERE fr=634562").show()
```

This post includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

1.7 Window functions and Pivot Tables with Pyspark

1.7.1 Resilient Distributed Datasets

Spark uses Java Virtual Machine (JVM) objects Resilient Distributed Datasets (RDD) which are calculated and stored in memory.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField,StringType,IntegerType,StructType,␣
↪DoubleType, FloatType
from pyspark.sql.functions import *


data_schema = [
StructField("_c0", IntegerType(), True)
,StructField("province", StringType(), True)
,StructField("specific", DoubleType(), True)
,StructField("general", DoubleType(), True)
,StructField("year", IntegerType(), True)
,StructField("gdp", FloatType(), True)
,StructField("fdi", FloatType(), True)
,StructField("rnr", DoubleType(), True)
,StructField("rr", FloatType(), True)
,StructField("i", FloatType(), True)
,StructField("fr", IntegerType(), True)
,StructField("reg", StringType(), True)
,StructField("it", IntegerType(), True)
]

final_struc = StructType(fields=data_schema)

file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").schema(final_struc).option("header", True).load(file_
↪location)

#df.printSchema()

df.show()
```

1.7.2 Using toPandas to look at the data

```
df.limit(10).toPandas()
```

1.7.3 Renaming Columns

```
df = df.withColumnRenamed("reg","region")
```

```
df.limit(10).toPandas()
```

```
# df = df.toDF(*['year', 'region', 'province', 'gdp', 'fdi', 'specific', 'general',
↪'it', 'fr', 'rnr', 'rr', 'i', '_c0', 'specific_classification', 'provinceIndex',
↪'regionIndex'])
```

1.7.4 Selecting Columns of Interest

```
df = df.select('year','region','province','gdp', 'fdi')
```

```
df.sort("gdp").show()
```

1.7.5 Sorting RDDs by Columns

```
from pyspark.sql import functions as F
df.sort(F.desc("gdp")).show()
```

1.7.6 Casting Data Types

```
from pyspark.sql.types import IntegerType, StringType, DoubleType
df = df.withColumn('gdp', F.col('gdp').cast(DoubleType()))
```

```
df = df.withColumn('province', F.col('province').cast(StringType()))
```

```
df.filter((df.gdp>10000) & (df.region=='East China')).show()
```

1.7.7 Aggregating using groupBy, .agg and sum/max

```
from pyspark.sql import functions as F

df.groupBy(["region","province"]).agg(F.sum("gdp") ,F.max("gdp")).show()
```

```
df.groupBy(["region","province"]).agg(F.sum("gdp").alias("SumGDP"),F.max("gdp").alias(
↪"MaxGDP")).show()
```

```
df.groupBy(["region","province"]).agg(
    F.sum("gdp").alias("SumGDP"),\
    F.max("gdp").alias("MaxGDP")\
    ).show()
```

```
df.limit(10).toPandas()
```

1.7.8 Exponentials using exp

```
df = df.withColumn("Exp_GDP", F.exp("gdp"))
df.show()
```

1.7.9 Window functions

Note: Window functions

```
# Window functions

from pyspark.sql.window import Window
windowSpec = Window().partitionBy(['province']).orderBy(F.desc('gdp'))
df.withColumn("rank",F.rank().over(windowSpec)).show()
```

```
from pyspark.sql.window import Window
windowSpec = Window().partitionBy(['province']).orderBy('year')
```

1.7.10 Lagging Variables

```
dfWithLag = df.withColumn("lag_7",F.lag("gdp", 7).over(windowSpec))
```

```
df.filter(df.year>'2000').show()
```

1.7.11 Looking at windows within the data

```
from pyspark.sql.window import Window

windowSpec = Window().partitionBy(['province']).orderBy('year').rowsBetween(-6,0)
```

```
dfWithRoll = df.withColumn("roll_7_confirmed",F.mean("gdp").over(windowSpec))
```

```
dfWithRoll.filter(dfWithLag.year>'2001').show()
```

```
from pyspark.sql.window import Window
windowSpec = Window().partitionBy(['province']).orderBy('year').rowsBetween(Window.
↪unboundedPreceding,Window.currentRow)
```

```
dfWithRoll = df.withColumn("cumulative_gdp",F.sum("gdp").over(windowSpec))
```

```
dfWithRoll.filter(dfWithLag.year>'1999').show()
```

1.7.12 Pivot Dataframes

Note: Pivot Dataframes

```
pivoted_df = df.groupBy('year').pivot('province') \
                      .agg(F.sum('gdp').alias('gdp') , F.sum('fdi').alias('fdi'))
pivoted_df.limit(10).toPandas()
```

```
pivoted_df.columns
```

```
newColnames = [x.replace("-","_") for x in pivoted_df.columns]
```

```
pivoted_df = pivoted_df.toDF(*newColnames)
```

```
expression = ""
cnt=0
for column in pivoted_df.columns:
    if column!='year':
        cnt +=1
        expression += f"'{column}' , {column},"

expression = f"stack({cnt}, {expression[:-1]}) as (Type,Value)"
```

1.7.13 Unpivoting RDDs

```
unpivoted_df = pivoted_df.select('year',F.expr(expression))
unpivoted_df.show()
```

This post includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

1.8 Regression and Classification with Pyspark ML

1.8.1 Linear Regression and Random Forest/GBT Classification with Pyspark

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField,StringType,IntegerType,StructType,␣
↪DoubleType, FloatType
from pyspark.sql.functions import *

data_schema = [
StructField("_c0", IntegerType(), True)
,StructField("province", StringType(), True)
,StructField("specific", DoubleType(), True)
,StructField("general", DoubleType(), True)
,StructField("year", IntegerType(), True)
,StructField("gdp", FloatType(), True)
,StructField("fdi", FloatType(), True)
,StructField("rnr", DoubleType(), True)
,StructField("rr", FloatType(), True)
,StructField("i", FloatType(), True)
,StructField("fr", IntegerType(), True)
,StructField("reg", StringType(), True)
,StructField("it", IntegerType(), True)
]

final_struc = StructType(fields=data_schema)

file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").schema(final_struc).option("header", True).load(file_
↪location)

#df.printSchema()
```

(continues on next page)

(continued from previous page)

```

df.show()
```

```
df.groupBy('province').count().show()
```

1.8.2 Imputation of mean values to prepare the data

```
mean_val = df.select(mean(df['general'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["general"])
```

```
mean_val = df.select(mean(df['specific'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["specific"])
```

```
mean_val = df.select(mean(df['rr'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["rr"])
```

```
mean_val = df.select(mean(df['fr'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["fr"])
```

```
mean_val = df.select(mean(df['rnr'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["rnr"])
```

```
mean_val = df.select(mean(df['i'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["i"])
```

1.8.3 Creating binary target feature from extant column for classification

```
from pyspark.sql.functions import *
df = df.withColumn('specific_classification',when(df.specific >= 583470.7303370787,1).
↪otherwise(0))
```

1.8.4 Using StringIndexer for categorical encoding of string type columns

```
from pyspark.ml.feature import StringIndexer
```

```
indexer = StringIndexer(inputCol="province", outputCol="provinceIndex")
df = indexer.fit(df).transform(df)
```

```
indexer = StringIndexer(inputCol="reg", outputCol="regionIndex")
df = indexer.fit(df).transform(df)
```

```
df.show()
```

1.8.5 Using VectorAssembler to prepare features for machine learning

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
```

```
df.columns
```

```
assembler = VectorAssembler(
 inputCols=[
 'provinceIndex',
# 'specific',
 'general',
 'year',
 'gdp',
 'fdi',
 #'rnr',
 #'rr',
 #'i',
 #'fr',
 'regionIndex',
 'it'
 ],
 outputCol="features")
```

```
output = assembler.transform(df)
```

```
final_data = output.select("features", "specific")
```

1.8.6 Spliting data into train and test

```
train_data,test_data = final_data.randomSplit([0.7,0.3])
```

1.8.7 Regression with Pyspark ML

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(labelCol='specific')
```

1.8.8 Fitting the linear regression model to the training data

```
lrModel = lr.fit(train_data)
```

1.8.9 Coefficients and Intercept of the linear regression model

```
print("Coefficients: {} Intercept: {}".format(lrModel.coefficients,lrModel.intercept))
```

1.8.10 Evaluating trained linear regression model on the test data

```
test_results = lrModel.evaluate(test_data)
```

1.8.11 Metrics of trained linear regression model on the test data (RMSE, MSE, R2)

```
print("RMSE: {}".format(test_results.rootMeanSquaredError))
print("MSE: {}".format(test_results.meanSquaredError))
print("R2: {}".format(test_results.r2))
```

1.8.12 Looking at correlations with corr

```
from pyspark.sql.functions import corr
```

```
df.select(corr('specific','gdp')).show()
```

1.8.13 Classification with Pyspark ML

```
from pyspark.ml.classification import DecisionTreeClassifier,GBTClassifier,
↪RandomForestClassifier
from pyspark.ml import Pipeline
```

1.8.14 DecisionTreeClassifier, RandomForestClassifier and GBTClassifier

```
dtc = DecisionTreeClassifier(labelCol='specific_classification',featuresCol='features
↪')
rfc = RandomForestClassifier(labelCol='specific_classification',featuresCol='features
↪')
gbt = GBTClassifier(labelCol='specific_classification',featuresCol='features')
```

1.8.15 Selecting features and binary target

```
final_data = output.select("features", "specific_classification")
train_data,test_data = final_data.randomSplit([0.7,0.3])
```

1.8.16 Fitting the Classifiers to the Training Data

```
rfc_model = rfc.fit(train_data)
gbt_model = gbt.fit(train_data)
dtc_model = dtc.fit(train_data)
```

1.8.17 Classifier predictions on test data

```
dtc_predictions = dtc_model.transform(test_data)
rfc_predictions = rfc_model.transform(test_data)
gbt_predictions = gbt_model.transform(test_data)
```

1.8.18 Evaluating Classifiers using pyspark.ml.evaluation and MulticlassClassificationEvaluator

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

Classifier Accuracy

```
acc_evaluator = MulticlassClassificationEvaluator(labelCol="specific_classification",␣
↪predictionCol="prediction", metricName="accuracy")
```

1.8.19 Classifier Accuracy Metrics

```
dtc_acc = acc_evaluator.evaluate(dtc_predictions)
rfc_acc = acc_evaluator.evaluate(rfc_predictions)
gbt_acc = acc_evaluator.evaluate(gbt_predictions)
```

```
print('-'*80)
print('Decision tree accuracy: {0:2.2f}%'.format(dtc_acc*100))
print('-'*80)
```

(continues on next page)

(continued from previous page)

```
print('Random forest ensemble accuracy: {0:2.2f}%'.format(rfc_acc*100))
print('-'*80)
print('GBT accuracy: {0:2.2f}%'.format(gbt_acc*100))
print('-'*80)
```

1.8.20 Classification Correlation with Corr

```
df.select(corr('specific_classification','fdi')).show()
```

```
df.select(corr('specific_classification','gdp')).show()
```

This post includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

1.9 Clustering with Pyspark

This post includes code from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField,StringType,IntegerType,StructType,␣
↪DoubleType, FloatType

data_schema = [
StructField("_c0", IntegerType(), True)
,StructField("province", StringType(), True)
,StructField("specific", DoubleType(), True)
,StructField("general", DoubleType(), True)
,StructField("year", IntegerType(), True)
,StructField("gdp", FloatType(), True)
,StructField("fdi", FloatType(), True)
,StructField("rnr", DoubleType(), True)
,StructField("rr", FloatType(), True)
,StructField("i", FloatType(), True)
,StructField("fr", IntegerType(), True)
,StructField("reg", StringType(), True)
,StructField("it", IntegerType(), True)
]

final_struc = StructType(fields=data_schema)

file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").schema(final_struc).option("header", True).load(file_
↪location)

#df.printSchema()

df.show()
```

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
```

```
feat_cols = ['year', 'gdp', 'fdi','fr']
```

```
feat_cols = ['gdp']
```

```
vec_assembler = VectorAssembler(inputCols = feat_cols, outputCol='features')
```

```
final_df = vec_assembler.transform(df)
```

```
from pyspark.ml.feature import StandardScaler
```

```
scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures", withStd=True,
↪ withMean=False)
```

```
# Compute summary statistics by fitting the StandardScaler
scalerModel = scaler.fit(final_df)
```

```
# Normalize each feature to have unit standard deviation.
cluster_final_data = scalerModel.transform(final_df)
```

```
kmeans3 = KMeans(featuresCol='scaledFeatures',k=3)
kmeans2 = KMeans(featuresCol='scaledFeatures',k=2)
```

```
model_k3 = kmeans3.fit(cluster_final_data)
model_k2 = kmeans2.fit(cluster_final_data)
```

```
model_k3.transform(cluster_final_data).groupBy('prediction').count().show()
```

```
model_k2.transform(cluster_final_data).groupBy('prediction').count().show()
```

1.10 NLP with Pyspark

This post includes code from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

```
from pyspark.ml.feature import Tokenizer, RegexTokenizer
from pyspark.sql.functions import col, udf
from pyspark.sql.types import IntegerType
```

```
sentenceDataFrame = spark.createDataFrame([
    (0, "Hi I heard about Spark"),
    (1, "I wish Java could use case classes"),
    (2, "Logistic,regression,models,are,neat")
], ["id", "sentence"])
```

```
sentenceDataFrame.show()
```

```
tokenizer = Tokenizer(inputCol="sentence", outputCol="words")

regexTokenizer = RegexTokenizer(inputCol="sentence", outputCol="words", pattern="\\W")
```

(continues on next page)

(continued from previous page)

```
# alternatively, pattern="\\w+", gaps(False)

countTokens = udf(lambda words: len(words), IntegerType())

tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("sentence", "words")\
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)

regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("sentence", "words") \
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)
```

```
## Stop Words Removal
```

```
from pyspark.ml.feature import StopWordsRemover

sentenceData = spark.createDataFrame([
    (0, ["I", "saw", "the", "red", "balloon"]),
    (1, ["Mary", "had", "a", "little", "lamb"])
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw", outputCol="filtered")
remover.transform(sentenceData).show(truncate=False)
```

```
## n-grams
```

```
from pyspark.ml.feature import NGram

wordDataFrame = spark.createDataFrame([
    (0, ["Hi", "I", "heard", "about", "Spark"]),
    (1, ["I", "wish", "Java", "could", "use", "case", "classes"]),
    (2, ["Logistic", "regression", "models", "are", "neat"])
], ["id", "words"])

ngram = NGram(n=2, inputCol="words", outputCol="ngrams")

ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.select("ngrams").show(truncate=False)
```

```
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame([
    (0.0, "Hi I heard about Spark"),
    (0.0, "I wish Java could use case classes"),
    (1.0, "Logistic regression models are neat")
], ["label", "sentence"])

sentenceData.show()
```

```
tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
wordsData = tokenizer.transform(sentenceData)
wordsData.show()
```

```
hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)
featurizedData = hashingTF.transform(wordsData)
# alternatively, CountVectorizer can also be used to get term frequency vectors

idf = IDF(inputCol="rawFeatures", outputCol="features")
idfModel = idf.fit(featurizedData)
rescaledData = idfModel.transform(featurizedData)

rescaledData.select("label", "features").show()
```

```
## CountVectorizer
```

```
from pyspark.ml.feature import CountVectorizer

# Input data: Each row is a bag of words with a ID.
df = spark.createDataFrame([
    (0, "a b c".split(" ")),
    (1, "a b b c a".split(" "))
], ["id", "words"])

# fit a CountVectorizerModel from the corpus.
cv = CountVectorizer(inputCol="words", outputCol="features", vocabSize=3, minDF=2.0)

model = cv.fit(df)

result = model.transform(df)
result.show(truncate=False)
```

```
df = spark.read.load("/FileStore/tables/SMSSpamCollection",
                     format="csv", sep="\t", inferSchema="true", header="false")
```

```
df.printSchema()
```

```
data = df.withColumnRenamed('_c0','class').withColumnRenamed('_c1','text')
```

```
data.printSchema()
```

```
## Clean and Prepare the Data
```

```
from pyspark.sql.functions import length
```

```
data = data.withColumn('length',length(data['text']))
```

```
data.printSchema()
```

```
# Pretty Clear Difference
data.groupby('class').mean().show()
```

```
from pyspark.ml.feature import Tokenizer,StopWordsRemover, CountVectorizer,IDF,
↪StringIndexer

tokenizer = Tokenizer(inputCol="text", outputCol="token_text")
stopremove = StopWordsRemover(inputCol='token_text',outputCol='stop_tokens')
```

(continues on next page)

(continued from previous page)

```
count_vec = CountVectorizer(inputCol='stop_tokens',outputCol='c_vec')
idf = IDF(inputCol="c_vec", outputCol="tf_idf")
ham_spam_to_num = StringIndexer(inputCol='class',outputCol='label')
```

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.linalg import Vector
```

```
clean_up = VectorAssembler(inputCols=['tf_idf','length'],outputCol='features')
```

```
### Naive Bayes
```

```
from pyspark.ml.classification import NaiveBayes
```

```
# Use defaults
nb = NaiveBayes()
```

```
### Pipeline
```

```
from pyspark.ml import Pipeline
```

```
data_prep_pipe = Pipeline(stages=[ham_spam_to_num,tokenizer,stopremove,count_vec,idf,
↪clean_up])
```

```
cleaner = data_prep_pipe.fit(data)
```

```
clean_data = cleaner.transform(data)
```

```
### Training and Evaluation
```

```
clean_data = clean_data.select(['label','features'])
```

```
clean_data.show()
```

```
(training,testing) = clean_data.randomSplit([0.7,0.3])
```

```
spam_predictor = nb.fit(training)
```

```
data.printSchema()
```

```
test_results = spam_predictor.transform(testing)
```

```
test_results.show()
```

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

```
acc_eval = MulticlassClassificationEvaluator()
acc = acc_eval.evaluate(test_results)
print("Accuracy of model at predicting spam was: {}".format(acc))
```

1.11 Regression and Classification with Pyspark ML

1.11.1 Linear Regression and Random Forest/GBT Classification with Pyspark

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField,StringType,IntegerType,StructType,␣
↪DoubleType, FloatType
from pyspark.sql.functions import *

data_schema = [
StructField("_c0", IntegerType(), True)
,StructField("province", StringType(), True)
,StructField("specific", DoubleType(), True)
,StructField("general", DoubleType(), True)
,StructField("year", IntegerType(), True)
,StructField("gdp", FloatType(), True)
,StructField("fdi", FloatType(), True)
,StructField("rnr", DoubleType(), True)
,StructField("rr", FloatType(), True)
,StructField("i", FloatType(), True)
,StructField("fr", IntegerType(), True)
,StructField("reg", StringType(), True)
,StructField("it", IntegerType(), True)
]

final_struc = StructType(fields=data_schema)

file_location = "/FileStore/tables/df_panel_fix.csv"
df = spark.read.format("CSV").schema(final_struc).option("header", True).load(file_
↪location)

#df.printSchema()

df.show()
```

```
df.groupBy('province').count().show()
```

1.11.2 Imputation of mean values to prepare the data

```
mean_val = df.select(mean(df['general'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["general"])
```

```
mean_val = df.select(mean(df['specific'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["specific"])
```

```
mean_val = df.select(mean(df['rr'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["rr"])
```

```
mean_val = df.select(mean(df['fr'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["fr"])
```

```
mean_val = df.select(mean(df['rnr'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["rnr"])
```

```
mean_val = df.select(mean(df['i'])).collect()
mean_val[0][0]
mean_gen = mean_val[0][0]
df = df.na.fill(mean_gen,["i"])
```

1.11.3 Creating binary target feature from extant column for classification

```
from pyspark.sql.functions import *
df = df.withColumn('specific_classification',when(df.specific >= 583470.7303370787,1).
↪otherwise(0))
```

1.11.4 Using StringIndexer for categorical encoding of string type columns

```
from pyspark.ml.feature import StringIndexer
```

```
indexer = StringIndexer(inputCol="province", outputCol="provinceIndex")
df = indexer.fit(df).transform(df)
```

```
indexer = StringIndexer(inputCol="reg", outputCol="regionIndex")
df = indexer.fit(df).transform(df)
```

```
df.show()
```

1.11.5 Using VectorAssembler to prepare features for machine learning

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
```

```
df.columns
```

```
assembler = VectorAssembler(
 inputCols=[
 'provinceIndex',
# 'specific',
 'general',
 'year',
 'gdp',
 'fdi',
```

(continues on next page)

(continued from previous page)

```
 #'rnr',
 #'rr',
 #'i',
 #'fr',
 'regionIndex',
 'it'
 ],
 outputCol="features")
```

```
output = assembler.transform(df)
```

```
final_data = output.select("features", "specific")
```

1.11.6 Spliting data into train and test

```
train_data,test_data = final_data.randomSplit([0.7,0.3])
```

1.11.7 Regression with Pyspark ML

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(labelCol='specific')
```

1.11.8 Fitting the linear regression model to the training data

```
lrModel = lr.fit(train_data)
```

1.11.9 Coefficients and Intercept of the linear regression model

```
print("Coefficients: {} Intercept: {}".format(lrModel.coefficients,lrModel.intercept))
```

1.11.10 Evaluating trained linear regression model on the test data

```
test_results = lrModel.evaluate(test_data)
```

1.11.11 Metrics of trained linear regression model on the test data (RMSE, MSE, R2)

```
print("RMSE: {}".format(test_results.rootMeanSquaredError))
print("MSE: {}".format(test_results.meanSquaredError))
print("R2: {}".format(test_results.r2))
```

1.11.12 Looking at correlations with corr

```
from pyspark.sql.functions import corr
```

```
df.select(corr('specific','gdp')).show()
```

1.11.13 Classification with Pyspark ML

```
from pyspark.ml.classification import DecisionTreeClassifier,GBTClassifier,
↪RandomForestClassifier
from pyspark.ml import Pipeline
```

1.11.14 DecisionTreeClassifier, RandomForestClassifier and GBTClassifier

```
dtc = DecisionTreeClassifier(labelCol='specific_classification',featuresCol='features
↪')
rfc = RandomForestClassifier(labelCol='specific_classification',featuresCol='features
↪')
gbt = GBTClassifier(labelCol='specific_classification',featuresCol='features')
```

1.11.15 Selecting features and binary target

```
final_data = output.select("features", "specific_classification")
train_data,test_data = final_data.randomSplit([0.7,0.3])
```

1.11.16 Fitting the Classifiers to the Training Data

```
rfc_model = rfc.fit(train_data)
gbt_model = gbt.fit(train_data)
dtc_model = dtc.fit(train_data)
```

1.11.17 Classifier predictions on test data

```
dtc_predictions = dtc_model.transform(test_data)
rfc_predictions = rfc_model.transform(test_data)
gbt_predictions = gbt_model.transform(test_data)
```

1.11.18 Evaluating Classifiers using pyspark.ml.evaluation and MulticlassClassificationEvaluator

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

Classifier Accuracy

```
acc_evaluator = MulticlassClassificationEvaluator(labelCol="specific_classification",␣
↪predictionCol="prediction", metricName="accuracy")
```

1.11.19 Classifier Accuracy Metrics

```
dtc_acc = acc_evaluator.evaluate(dtc_predictions)
rfc_acc = acc_evaluator.evaluate(rfc_predictions)
gbt_acc = acc_evaluator.evaluate(gbt_predictions)
```

```
print('-'*80)
print('Decision tree accuracy: {0:2.2f}%'.format(dtc_acc*100))
print('-'*80)
print('Random forest ensemble accuracy: {0:2.2f}%'.format(rfc_acc*100))
print('-'*80)
print('GBT accuracy: {0:2.2f}%'.format(gbt_acc*100))
print('-'*80)
```

1.11.20 Classification Correlation with Corr

```
df.select(corr('specific_classification','fdi')).show()
```

```
df.select(corr('specific_classification','gdp')).show()
```

This post includes code adapted from Spark and Python for Big Data udemy course and Spark and Python for Big Data notebooks.

BIBLIOGRAPHY

[Kea19] David Raymond Kearney. *Ties that Bind: Connections, Institutions and Economics in the People's Republic of China*. PhD thesis, Duke University, 2019.