



TUTORIAL PYSPARK E MLLIB

10 Minutos de leitura



TUTORIAL DE PYSPARK E MLLIB

Nesse tutorial de Spark vamos utilizar PySpark e MLLib para uma atividade simples de processamento de [Machine Learning](#).

UMA BREVE INTRODUÇÃO AO HADOOP E SPARK

Com o advento do [Big Data](#), faz-se necessário inserir novas técnicas de



Neste cenário, 2 ferramentas se destacam no mercado. São elas, o ecossistema [Hadoop](#), e Spark.

Hadoop combina diversas ferramentas para armazenamento de dados e queries, como Hive, Pig, MapReduce, entre outras. Já o Spark surgiu como uma alternativa para o MapReduce do Hadoop, mas tem-se provado uma excelente ferramenta para realização de computação distribuída, ou seja, processamento paralelo entre diversos nós de um cluster de computadores.

INICIANDO O TUTORIAL DE PYSPARK

O objetivo deste artigo é a exploração da API do spark para Python, o Pyspark, e da biblioteca de [Machine Learning](#), MLib, para realizar a análise exploratória de um conjunto de dados e fazer uma Regressão Linear.

Vemos o PySpark como um excelente alternativa para programação em Spark, já que une uma linguagem que está se tornando muito popular : o [Python](#) e o [Spark](#)

IMPORTE O FINDSPARK

Primeiro, deve-se importar o módulo findspark de modo à utilizar o método .init, responsável por inicializar o spark.

In [1]:

```
# Import findspark
import findspark
```

```
# Initialize and provide path
```



```
IMPORT SPARKSESSION
```

Após o spark ter sido encontrado no sistema, é necessário criar a Sessão Spark, onde é possível configurar os nós do cluster, bem como a memória alocada para cada um deles.

In [2]:

```
# Import SparkSession
from pyspark.sql import SparkSession

# Build the SparkSession
spark = SparkSession.builder \
    .master("local") \
    .appName("Linear Regression Model") \
    .config("spark.executor.memory", "1gb") \
    .getOrCreate()

sc = spark.sparkContext
```

INICIANDO O DESENVOLVIMENTO COM SPARK

Com a sessão spark criada, pode-se trabalhar no ambiente de desenvolvimento. A primeira etapa é importar o conjunto de dados, neste caso, o arquivo chama-se “Salary_Data.csv”, contendo dados de determinados funcionários, com os seus salários e anos de experiência em determinada função.

Note que os dados foram salvos em uma variável chamada `rdd`, que significa



Desta forma pode-se processá-los em paralelo, aumentando a velocidade de processamento.

In [3]:

```
rdd = sc.textFile('<path-to-data>/Salary_Data.csv')
```

O método `.take` é indicado para visualização de uma parcela dos dados. Neste caso, o retorno da função trás 2 entradas do dataset. Note que a primeira entrada é '1.1,39343.00' e a segunda entrada é '1.3,46205.00'. Os valores 1.1 e 1.3 representam os anos de experiência de um funcionário, já as entradas 39343.00 e 46205.00 representam os seus respectivos salários.

In [4]:

```
rdd.take(2)
```

Out[4]:

```
['1.1,39343.00', '1.3,46205.00']
```

Como as entradas vieram juntas na mesma string, é necessário separar os valores pela vírgula. Deste modo, usa-se o método `split(",")`, responsável por isso. Também usa-se a função `map`, que mapeia a operação entre parênteses para todas as linhas do `rdd`.

Note que foi usado paradigma funcional de programação com a função `lambda line: line.split(",")`. O Spark se beneficia deste paradigma, portanto é necessário utilizá-lo.

In [5]:



```
# Inspect the first line  
rdd.take(1)
```

Out[5]:

```
[[ '1.1', '39343.00' ]]
```

Observe que os valores foram separados pela vírgula, como esperado.

Existem os métodos `.first` e `.top`, que mostram a primeira linha, e a linha do topo, respectivamente. São métodos semelhantes.

In [6]:

```
# Inspect the first line  
rdd.first()
```

Out[6]:

```
[ '1.1', '39343.00' ]
```

In [7]:

```
# Take top elements  
rdd.top(1)
```

Out[7]:

```
[[ '9.6', '112635.00' ]]
```

Nesta etapa, importa-se o módulo Row, onde o rdd faz a transformação



Mapeou-se todas as linhas para a formatação de colunas especificadas, e chamou-se o método `.toDF()`, onde é feita a transformação do rdd para DataFrame (semelhante ao DataFrame da biblioteca pandas).

In [8]:

```
# Import the necessary modules
from pyspark.sql import Row

# Map the RDD to a DF
df = rdd.map(lambda line: Row(YearsExperience=line[0],
Salary=line[1])).toDF()
```

Usando o método `.show`, é possível inspecionar como o DataFrame está.

In [9]:

```
# Show the top 20 rows
df.show()
```

```
+-----+-----+
|  Salary|YearsExperience|
+-----+-----+
|39343.00|           1.1|
|46205.00|           1.3|
|37731.00|           1.5|
|43525.00|           2.0|
|39891.00|           2.2|
```

```
|56642.00|      3.0|
```



```
|57189.00|      3.7|
|63218.00|      3.9|
|55794.00|      4.0|
|56957.00|      4.0|
|57081.00|      4.1|
|61111.00|      4.5|
|67938.00|      4.9|
|66029.00|      5.1|
|83088.00|      5.3|
|81363.00|      5.9|
|93940.00|      6.0|
```

```
+-----+-----+
```

only showing top 20 rows

O método `.printSchema` mostra algumas informações sobre os tipos de dados presentes nas colunas, conforme linha abaixo.

In [10]:

```
df.printSchema()
```

```
root
```

```
 |-- Salary: string (nullable = true)
```

```
 |-- YearsExperience: string (nullable = true)
```

Criou-se uma função chamada `convertColumn` que recebe como argumento



Logo após a criação da função, define-se a variável `columns`, como uma lista contendo os nomes das colunas do `df`, e aplica-se a função para o dataframe em si, convertendo os valores para `FloatType`.

In [11]:

```
# Import all from `sql.types`
from pyspark.sql.types import *

# Write a custom function to convert the data type of
DataFrame columns
def convertColumn(df, names, newType):
    for name in names:
        df = df.withColumn(name, df[name].cast(newType))
    return df

# Assign all column names to `columns`
columns = ['YearsExperience', 'Salary']

# Conver the `df` columns to `FloatType()`
df = convertColumn(df, columns, FloatType())
```

In [12]:

```
df.show()
```

```
+---+-----+
```

```
| Salary|YearsExperience|
```

```
+-----+-----+
```



```
| 30342.0 | 1.1 |
```



```
| 39891.0 | 2.2 |
```

```
| 56642.0 | 2.9 |
```

```
| 60150.0 | 3.0 |
```

```
| 54445.0 | 3.2 |
```

```
| 64445.0 | 3.2 |
```

```
| 57189.0 | 3.7 |
```

```
| 63218.0 | 3.9 |
```

```
| 55794.0 | 4.0 |
```

```
| 56957.0 | 4.0 |
```

```
| 57081.0 | 4.1 |
```

```
| 61111.0 | 4.5 |
```

```
| 67938.0 | 4.9 |
```

```
| 66029.0 | 5.1 |
```

```
| 83088.0 | 5.3 |
```

```
| 81363.0 | 5.9 |
```

```
| 93940.0 | 6.0 |
```

```
+-----+-----+
```

```
only showing top 20 rows
```

Também é possível mostrar apenas uma coluna com o método `.select`.

In [13]:

```
df.select('Salary').show(10)
```

```
+-----+
```

```
| Salary|
```



```
|43525.0|
```

```
|39891.0|
```

```
|56642.0|
```

```
|60150.0|
```

```
|54445.0|
```

```
|64445.0|
```

```
|57189.0|
```

```
+-----+
```

only showing top 10 rows

Outra operação bastante conhecida é o groupby, onde pode-se agrupar os dados por um determinado pivô. Neste caso, usa-se a coluna Salary como pivô, efetuando a contagem dos valores e ordenando-os em ordem decrescente.

In [14]:

```
df.groupBy("Salary").count().sort("Salary",ascending=False).show()
```



```
+-----+-----+
```

```
|  Salary|count|
```

```
+-----+-----+
```

```
|122391.0|    1|
```

```
|121872.0|    1|
```

```
|116969.0|    1|
```

```
|113812.0|    1|
```

```
| 112625.0 | 1 |
```



```
| 98273.0 | 1 |
```

```
| 93940.0 | 1 |
```

```
| 91738.0 | 1 |
```

```
| 83088.0 | 1 |
```

```
| 81363.0 | 1 |
```

```
| 67938.0 | 1 |
```

```
| 66029.0 | 1 |
```

```
| 64445.0 | 1 |
```

```
| 63218.0 | 1 |
```

```
| 61111.0 | 1 |
```

```
| 60150.0 | 1 |
```

```
| 57189.0 | 1 |
```

```
+-----+-----+
```

only showing top 20 rows

Por último, temos o método `.describe`, que faz a descrição do df baseado nas colunas, retornando uma contagem dos elementos, a média, desvio padrão, valores mínimo e máximo.

In [15]:

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

```
+-----+-----+-----+
|summary|          Salary|  YearsExperience|
+-----+-----+-----+
|  count|             30|             30|
```

```
| mean | 76002.0 | 5.2122222642277485 |
```



```
+-----+-----+-----+-----+
```

Agora é hora de começar a tratar os dados de modo a deixá-los no formato que o algoritmo de [Machine Learning](#) espera. Para isso, usa-se o módulo `DenseVector`. Este `DenseVector` é uma maneira otimizada de lidar com valores numéricos, acelerando o processamento realizado pelo Spark.

Assim, mapeia-se as linhas do `df` transformando-as em `DenseVector`, e cria-se um novo dataframe, chamado `df`, com as colunas 'label' e 'features'.

Recordando que uma Regressão Linear é um problema de Aprendizado Supervisionado, ou seja, o algoritmo necessita do 'ground truth', os rótulos das entradas, de modo que ele possa comparar com sua saída e calcular alguma métrica de erro, como Erro Quadrático Médio (do inglês, Mean Squared Error, MSE), bastante empregado em problemas de Regressão.

In [16]:

```
# Import `DenseVector`
from pyspark.ml.linalg import DenseVector

# Define the `input_data`
input_data = df.rdd.map(lambda x: (x[0], DenseVector(x[1:])))

# Replace `df` with the new DataFrame
df = spark.createDataFrame(input_data, ["label", "features"])
```

Outra etapa importante é deixar os dados na mesma escala. Isso se faz necessário pelo fato de que o algoritmo de Regressão Linear trabalha com

distâncias euclidianas, ou seja, ele realiza operações de distância entre



Para isso, usou-se o método de normalização conhecido como StandardScaler, onde subtrai-se a média do valor x definido, e divide-se pela diverença ($x_{\max} - x_{\min}$). Desta forma, os dados estarão distribuídos ao longo de 0 na mesma escala.

Realiza-se o fit e o transform em cima do df, desta forma a variável scaled_df contém nosso label, nossas features, e nossas features já escaladas, conforme output desta célula.

In [17]:

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

# Initialize the `standardScaler`
standardScaler = StandardScaler(inputCol="features",
                                outputCol="features_scaled")

# Fit the DataFrame to the scaler
scaler = standardScaler.fit(df)

# Transform the data in `df` with the scaler
scaled_df = scaler.transform(df)

# Inspect the result
scaled_df.take(2)
```

Out[17]:



```
features_scaled=DenseVector([0.4581]))]
```

Aqui é o ponto onde o [Machine Learning](#) começa. Como primeira etapa, é necessário dividir nosso conjunto de dados em treino e teste. Para isso, divide-se de forma aleatória com tamanhos 75% para treino e 25% para teste, com seed 1234.

Essa etapa é necessária, porque o principal objetivo para o algoritmo de [Machine Learning](#) é que ele tenha capacidade de generalização, ou seja, consiga generalizar bem, obtendo boas métricas para dados não presentes na etapa de treinamento. Desta forma, o workflow desejado seria que o algoritmo fosse treinado em dados conhecidos (conjunto de treino), e atingisse boas métricas para o conjunto de testes (dados nunca antes vistos pelo algoritmo).

O elemento seed 1234 insere um elemento de randomização padrão, ou seja, a divisão de treino e testes será feita de maneira aleatória, porém, sempre terá o mesmo resultado para o mesmo seed. Isso é importante para reprodução de resultados.

In [18]:

```
# Machine Learning Begins

# Split the data into train and test sets
train_data, test_data =
scaled_df.randomSplit([.75,.25],seed=1234)
```

Importa-se o módulo LinearRegression, e instancia-se a classe com o objeto lr. Deve-se passar os parâmetros labelCol='label' sendo estes os labels, ou

rótulos do nosso problema de aprendizado supervisionado. O parâmetro



Criado o objeto da classe `LinearRegression`, pode-se aplicar o método `fit`, que é responsável pelo treinamento do algoritmo no `train_data`, conjunto de treino.

In [19]:

```
# Import `LinearRegression`
from pyspark.ml.regression import LinearRegression

# Initialize `lr`
lr = LinearRegression(labelCol="label", maxIter=10)

# Fit the data to the model
linearModel = lr.fit(train_data)
```

Cria-se a variável `predicted`, que é a predição do algoritmo para o conjunto de testes (`test_data`).

É realizada a extração de `predictions` da variável `predicted`, e também a extração dos `labels`, de modo que se possa compará-los lado a lado na variável `predictionAndLabel`.

Essa variável contém ambas as predições e os rótulos verdadeiros.

In [20]:

```
# Generate predictions
predicted = linearModel.transform(test_data)

# Extract the predictions and the "known" correct labels
predictions = predicted.select("prediction").rdd.map(lambda x:
```

```
predictionAndLabel = predictions.zip(labels).collect()

# Print out first 5 instances of `predictionAndLabel`
predictionAndLabel[:5]
```

Out[20]:

```
[(40930.199755811234, 37731.0),
 (37273.287820618236, 39343.0),
 (47329.79645978254, 39891.0),
 (62871.67354665871, 63218.0),
 (102183.47657752226, 113812.0)]
```

Pode-se extrair os coeficientes da equação da reta:

$$y = a_0 \cdot x + a_1$$

Onde coefficients é o valor de a_0 , e intercept é o valor de a_1 .

In [21]:

```
# Coefficients for the model
linearModel.coefficients
```

Out[21]:

```
DenseVector([9142.2804])
```

In [22]:



Out[22]:

```
27216.779181453927
```

Também é possível extrair o Erro Quadrático Médio, em inglês MSE, neste caso sendo representado pela raiz quadrada deste valor(RMSE).

In [23]:

```
# Get the RMSE
linearModel.summary.rootMeanSquaredError
```

Out[23]:

```
5582.542549720365
```

E como métrica de avaliação do modelo, extrai-se o R2, ou Coeficiente de Determinação, uma métrica estatística de proximidade de pontos e reta sobreposta.

In [24]:

```
# Get the R2
linearModel.summary.r2
```

Out[24]:

```
0.9477919736672404
```

In [25]:



ENCERRANDO TUTORIAL

Esperamos ter ajudado com uma introdução simples sobre PySpark, MLLib e talvez até mesmo o primeiro contato com Spark

Arquivos para Download

[Salary Data](#)



SOBRE O AUTOR



CETAX

Somos uma consultoria de Business Intelligence e Data Warehousing que atua desde 2000, guiando as empresas a transformar seus dados em valiosas informações que transformam os seus negócios.

VER TODOS OS

POSTS





março 22, 2018

COMO CRIAR SEU PRIMEIRO DATAFLOW COM APACHE NIFI

por: Cetax / 6 Minutos de leitura / 1 Comentário



Tutorial usando Apache Nifi Nesse artigo vamos mostrar de uma maneira simples como criar um data flow para fazer integração de dados. O Apache Nifi é um projeto Open Source de integração de dados, que integra diversas origens com diversos tipos de destinos, usando bancos de dados, Hadoop (HDFS), Kafka, Spark, entre outros. Conheça mais sobre o Apache Nifi na página do Projeto : Preparando...

março 22, 2018

por: Cetax / 3 Minutos de leitura



O que é o Apache Phoenix e para que serve? O Apache Phoenix é desenvolvido em java para rodar uma camada SQL para ser interpretada em banco de dados NoSQL. Ele permite que os usuários criem, excluam, alterem tabelas, visualizações, índices, sequências, Insira e exclua as linhas individualmente e em massa e dados de consulta através de uma camada SQL sobre o HBase. Apache Phoenix É performático? O...

/*]]> */