Daniel dos Santos

Análise de dados de alta dimensão utilizando Apache Spark com R

Niterói - RJ, Brasil

13 de setembro de 2021

Universidade Federal Fluminense

Daniel dos Santos

Análise de dados de alta dimensão utilizando Apache Spark com R

Trabalho de Conclusão de Curso

Monografia apresentada para obtenção do grau de Bacharel em Estatística pela Universidade Federal Fluminense.

Orientador(a): Prof. Dr. Douglas Rodrigues Pinto

Niterói - RJ, Brasil

13 de setembro de 2021

Universidade Federal Fluminense

Daniel dos Santos

Análise de dados de alta dimensão utilizando Apache Spark com R

Monografia de Projeto Final de Graduação sob o título *"Análise de dados de alta dimensão utilizando Apache Spark com R"*, defendida por Daniel dos Santos e aprovada em 13 de setembro de 2021, na cidade de Niterói, no Estado do Rio de Janeiro, pela banca examinadora constituída pelos professores:

Prof. Dr. Douglas Rodrigues Pinto
Departamento de Estatística – UFF

Profa. Dra. Karina Yuriko Yaginuma
Departamento de Estatística – UFF

Profa. Dra. Jessica Quintanilha Kubrusly
Departamento de Estatística – UFF

Niterói, 13 de setembro de 2021

Resumo

Desde o começo da Terceira Revolução Industrial, o volume de dados armazenados cresce exponencialmente, marcando este período como a Era da Informação. A capacidade de explorar tamanha quantidade de dados abre oportunidades para novas formas de análise e descobertas. Com o intuito de realizar tais análises de larga escala foi desenvolvido o Apache Spark, um framework de código aberto que busca democratizar estudos com dados de alta dimensão, utilizando técnicas de computação distribuída já fornecidas pelo MapReduce, porém com grandes melhorias em performance e praticidade. O Spark possui uma série de componentes que envolvem aprendizado de máquinas, análise de grafos, processamento de dados em tempo real e a realização de análises estatísticas em grandes volumes de dados. O intuito deste trabalho é explicar, apresentar e explorar a gama de ferramentas encontradas no Spark, utilizando-se das tecnologias e arquiteturas encontradas nele em conjunto com a linguagem de programação R a partir da biblioteca SparkR.

Palavras-chave: Apache Spark. Big data. Computação distribuída. Engenharia de dados. MapReduce. R.

Dedicatória

Dedico esta trabalho aos meus pais e minhas avós, que sempre estiveram ao meu lado prontos para me apoiar nos momentos mais difíceis. Também dedico aos meus professores e amigos que conheci durante a graduação.

Agradecimentos

Agradeço ao meu orientador o Prof. Dr. Douglas Rodrigues Pinto por toda a paciência e dedicação para me auxiliar neste projeto e em todos os outros que tive o prazer de confeccionar sob sua tutoria, obrigado pela confiança. Também agradeço a banca avaliadora pela disponibilização em participar e em ler atentamente este trabalho. Aos meus amigos que estiveram ao meu lado durante a graduação, Gabriel Mizuno, Letícia Felix, Luiz Fernando, Lyncoln Sousa e Rodolfo Hauret e aos meus amigos do grupo TCI, muito obrigado.

Conteúdo

Lista de Figuras

Glossário

API - Application Programming Interface: Interface de Programação de Aplicações;

Bins (Gráficos - Histogramas): Número de barras;

Camel Case: Modelo de escrita geralmente utilizado em códigos na qual não utiliza-se espaços para separação de palavras, mas letras maiúscula, como por exemplo: *CamelCase*;

Cluster: Sistemas agregados que trabalham em conjunto. São uma coleção de *Nodes*;

Crawler: Robô programado para acessar sites diversas vezes e "raspar"informações diretamente das páginas;

CSV - Comma-separated values: Valores separados por vírgula;

Dashboard: Painel que contém informações sobre o conjunto de dados. Podem incluir, tabelas, gráficos e outros tipos de visualização;

Framework: É uma abstração que incorpora um conjunto de funcionalidades que facilitam a escrita de códigos para determinadas finalidades;

IoT - Internet of Things: Internet das Coisas

JVM - Java Virtual Machine: Máquina Virtual Java;

MLP - Multilayer Perceptron: Perceptron Multicamadas;

Nodes: É o sistema individual que pertence ao um *Cluster*;

Pipelines: Na programação *pipeline*, ou em português dutos, são o conjunto de transformações realizadas nos dados de entrada para que tenha-se a saída desejada;

Query: É a "pergunta"a ser feita a um banco de dados, ou seja, o código responsável por realizar a consulta.

Queries: Plural de *Query*.

RDD - Resilient Distributed Dataset: Conjunto de dados distribuído e resiliente.

SQL - Standard Query Language: Linguagem de Consulta Padrão.

Streaming: Transmissão em tempo real.

Worker: *Node* responsável por relizar tarefas dentro de um *cluster*.

1 Introdução

1.1 Motivação

A necessidade de processar e analisar dados com confiabilidade, velocidade e escalabilidade, cresce a medida que tecnologia avança. Para suprir estas necessidades foi utilizado o MapReduce, um paradigma computacional capaz de fornecer um ambiente incrivelmente escalável, rápido e flexível. Porém, este método possuí algumas limitações que foram superadas através do Spark, um framework de código aberto, que atualmente é amplamente utilizado mundialmente pelas mais diversas organizações. Assim, a habilidade de lidar com o Spark para processar e análisar um grande volume de dados, até mesmo em tempo real, tornou-se uma característica desejável para profissionais que almejam atuar neste campo.

1.2 Revisão Bibliográfica

Na confecção deste trabalho foram utilizadas as documentações oficiais da linguagem de programação R e do framework Apache Spark. Também revisou-se o trabalho responsável pela concepção do Apache Spark, além de uma série de artigos que comparavam Spark ao Hadoop MapReduce em diversas situações.

1.3 Objetivos

O objetivo desta monografia é conduzir o leitor a conhecer sobre as arquiteturas utilizadas para processar dados de alta dimensão, e assim explorar as funcionalidades do Spark e sua integração com a linguagem R. Por fim, realizar uma comparação que verifica a eficiência do Spark frente aos métodos de análise tradicionais.

1.4 Organização

A organização deste texto busca a máxima reprodutibilidade do que é apresentado, por isso, é extremamente recomendável que o leitor utilize-se desta monografia na presença de um computador, com o intuito de aplicar durante a leitura os tópicos aqui mostrados. Esta monografia possui um repositório no qual podem ser acessados os códigos e modelos treinados. O repositório pode ser acessado no seguinte endereço <https://github.com/Daniel-EST/spark-tcc>.

2 Materiais e Métodos

2.1 Big data

Avanços tecnológicos, a ascensão do *IoT* e a preocupação em entender fenômenos e comportamentos, marcam o século XXI como a era orientada a dados. Uma simples troca de e-mails, assistir a um seriado televisivo ou navegar pela internet é o suficiente para que um grande número de informações sejam disparadas para os mais diversos bancos de dados mundiais, onde lá serão processados e servirão de insumo para algorítimos de recomendação, inteligência artificial, entre outros. Com esta quantidade infinita de dados sendo gerada a cada segundo de inúmeras fontes, tornou-se um desafio do terceiro milênio, transformar toda esta informação em respostas, soluções e melhorias na qualidade de vida de toda sociedade, isto marca o *big data*.

Douglas Laney em 2001 formulou uma das mais respeitadas definições de *big data* utlizando-se de um artifício nomeado de os "V's do *big data*", sendo eles:

- **Volume**: A cada momento a quantidade de dados a serem processados aumenta de forma significativa;
- **Velocidade**: As informações são passadas de forma muito rápida, como dados em *streaming*;
- **Variedade**: Dados provenientes de inúmeras fontes e formatos.

Assim, foram desenvolvidas uma série de ferramentas como o Hadoop e o Spark, tema desta monografia, que auxliam na manipulação e análise destes dados complexos, tornando-se essenciais para o confecção das mais diversas soluções envolvendo *big data*.

2.2 Spark

Com o desenvolvimento iniciado em 2009 em Berkley, Universidade da Califórnia, posteriormente tornando-se de código aberto (2010) e por fim doado à fundação Apache (2013), o Spark é uma ferramenta de uso geral para computação baseada em *cluster*. Sua capacidade em lidar com grandes volumes de dados utilizando computação paralelizada, em memória e tolerante à falhas, tornou-o referênica neste campo de estudos.

Figura 1: Logomarca do Apache Spark

Além de sua eficiência, o que torna o Spark popular é a facilidade em aplicá-lo. Apesar de ter sido escrito primeiramente em Scala, foram desenvolvidas *APIs* de alto nível para linguagens de programação como, Scala, Java, Python e R, permitindo a integração de grande parte de suas funcionalidades aos mais diferentes tipos de projeto. Assim, o Spark é adotado por diversas empresas no mundo, independente do tipo de indústria, pode-se citar empresas como Yahoo!, Huwaei, IBM e a Tencent, empresa chinesa da indústria de jogos eletrônicos, onde o Spark coordena mais de 8000 *nodes*. Já no Alibaba, empresa chinesa especializada em comércio online, o Spark foi responsável por lidar com mais de 1PB (petabyte) de informação, um marco em sua história.

2.3 MapReduce

MapReduce é o nome dado a implementação de um código que possui duas etapas principais. A primeira consiste em processar um conjunto de valores e transformá-los em valores intermediários, esta etapa é comumente nomeada de "Map". A etapa final é processar este valores intermediários e resumi-los, de forma que gerem o resultado esperado, sendo chamada de "Reduce". Suponha-se que deseja-se calcular a quantidade de valores maiores do que 5 em um conjunto de dados. A etapa Map seria responsável por modificar os dados de entrada para um valores intermediários. Neste caso, 1 para valores acima de 5 e 0 (zero) caso contrário. Por fim, a etapa Reduce somaria os resultados anteriores, gerando o número total de valores acima de 5, conforme o diagrama abaixo.

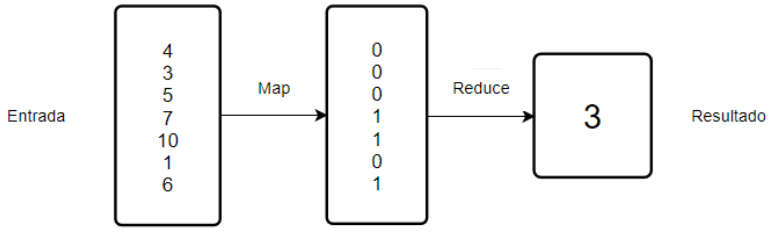

Figura 2: Exemplo da aplicação de MapReduce

Esta técnica é facilmente escalável, já que é possível que as etapas sejam paralelizadas entre diversas máquinas, apenas adicionando um passo anterior que separa e distribui os dados entre elas. Cada uma será responsável por aplicar o MapReduce em uma parcela dos dados e por fim consolidando em um resultado final. Como no diagrama abaixo:

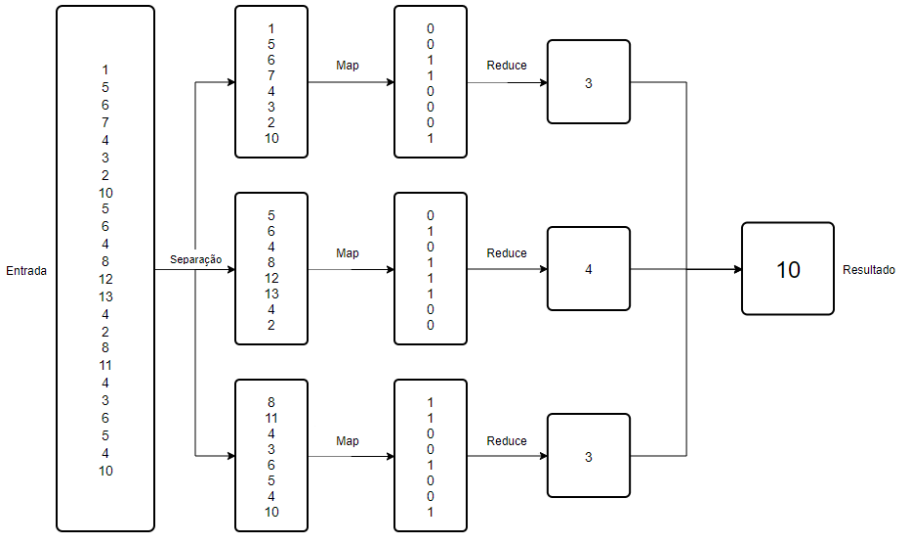

Figura 3: Exemplo da aplicação de MapReduce paralelizada

Porém, o MapReduce ocorre de forma acíclica, ou seja, após a execução do algorítimo os novos dados são salvos novamente em disco, o que gera lentidão em caso de tarefas iterativas, que necessitem passar diversas vezes pelo passo de MapReduce, como por exemplo, a aplicação de algorítimos de aprendizado de máquina. A solução apresentada pelo Spark foram os RDDs, que permitem que diversas tarefas sejam realizadas sem a necessidade de salvar em disco.

2.4 Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets (Conjunto de dados distribuído e resiliente) ou RDDs é a estrutura de somente leitura do Spark que permite com que tarefas sejam paralelizadas

e processadas em memória, esta foi a solução encontrada para superar as limitações do MapReduce e agilizar ainda mais as tarefas. Sem a necessidade de salvar em disco toda vez que uma transformação nos dados é executada, os RDDs conseguem a cada iteração, manter as informações em memória, isto se transforma em uma grande vantagem em eficiência em comparação ao uso tradicional do Hadoop MapReduce.

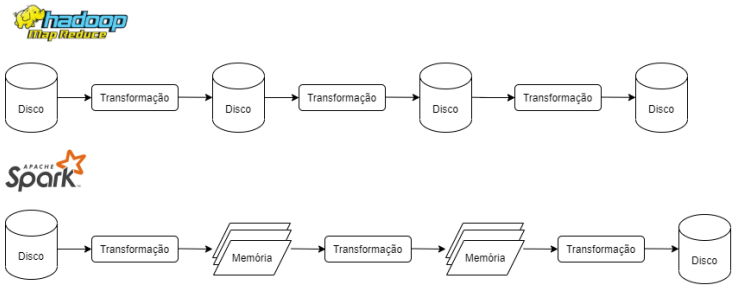

Figura 4: Comparação entre o ciclo do Hadoop MapReduce e o Spark

Atualmente, os RDDs são consideradas uma componente de baixo nível, na qual os DataFrames e Datasets foram escritos tendo-os como base.

2.5 Tolerância à falhas

Tolerância à falhas é a característica que define a capacidade de um sistema se recuperar e não perder informações, mesmo após a ocorrência de problemas no fluxo. Esta característica é extremamente desejável para análise dados, principalmente em tempo real.

No esquema simplificado abaixo supõe-se que as tarefas foram divididas entre três *workers*.

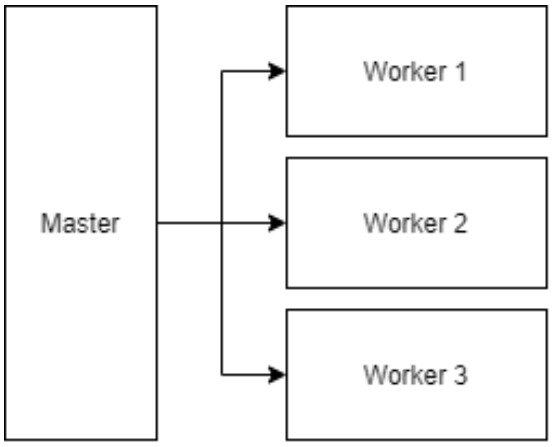

Figura 5: Esquema simplificado de um sistema distribuiído

Após algum tempo o *worker* 3 passa a apresentar falhas.

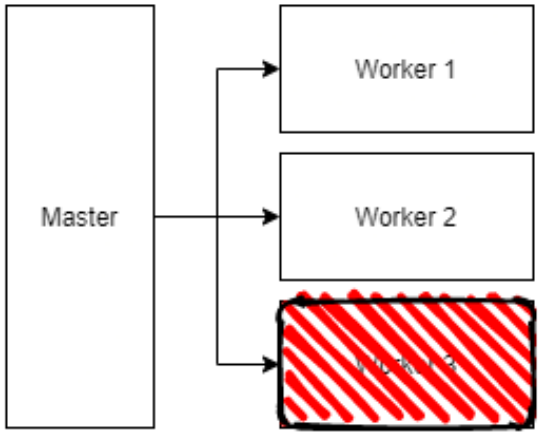

Figura 6: Esquema simplificado de um sistema distribuiído após falha em um dos *workers*

Neste caso, as tarefas serão repassadas aos outros *workers* sem perda de informação, porém isto não impede que possam ocorrer perdas de eficiência.

2.6 O Ecossistema Spark

O Ecossistema do Spark pode ser dividido em 5 partes: Spark Core, Spark SQL, Spark Streaming, Spark MLlib e Spark GraphX. Onde o Spark Core é a principal parte e as outas são bibliotecas de alto nível criadas com o intuito de unificar diversas partes de fluxos complexos de arquitetura.

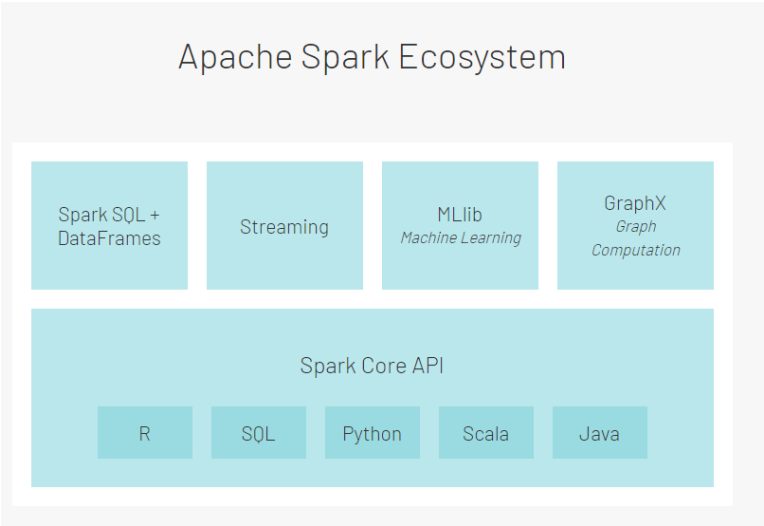

Figura 7: Ecossistema do Apache Spark

2.6.1 Spark Core

O Spark Core é o seu núcleo, a parte mais importante do ecossistema, que permitiu que outras componentes fossem construídas sobre ele. Sua interface disponibiliza as funções básicas e toda a arquitetura necessária para o processamento de grandes volumes de dados. É no Spark Core, que se encontram os RDDs, por exemplo.

2.6.2 Spark SQL

Para facilitar o processamento de dados estruturados, o Spark SQL foi desenvolvido. Com ele, é possível executar *queries* em SQL, ou seja, consultar bancos de dados utilizando uma linguagem de consulta padrão amplamente utilizada, melhorando ainda mais a forma de trabalhar com esses tipos de dados. Além disso, ao executar uma *query* pelo Spark SQL em uma linguagem de programação como o Python ou R, seu resultado será do tipo DataFrame, porém com algumas diferenças.

DataFrames para o Spark são conceitualmente equivalentes aos data frames encontrados no Python e no R, sua diferença é que internamente existem otimizações e outras ferramentas que permitem que sejam mais eficientes.

2.6.3 Spark Streaming

Esta extensão do Spark Core, permite o processamento de dados em *streaming*, ou seja, em tempo real, oferecendo escabilidade, alto rendimento e tolerância à falhas[1], características essenciais para este tipo de projeto. Suponha que, deseja-se construir um *dashboard* que atualiza seus dados em tempo real, para isso, basta conectar a fonte dos dados ao Spark Streaming, realizar o processamento necessário e lançá-los ao *dashboard*, como na figura abaixo.

Figura 8: Diagrama simplificado de processamento de dados em tempo real

[1]Tolerância à falhas, é a característica que permite que sistemas continuem operando mesmo após a ocorrência de alguma instabilidade que resulte na falha de alguma parte do processo.

2.6.4 Spark MLlib

Spark MLlib ou Spark Machine Learning Library tem como objetivo trazer toda praticidade e velocidade do Spark para a aplicação de aprendizado de máquina de forma escalável. Com ele é possível realizar o treinamento de diversos modelos de regressão e classificação, também é com esta componente que possibilita calcular diversas estatísticas dos dados e a realização de alguns testes de hipóteses.

2.6.5 Spark GraphX

GraphX é a componente que possuí diversas ferramentas para o processamento e análise de grafos complexos. Utilizando os *RDDs* como base, o GraphX estende suas funcionalidades, adicionando abstrações que permitem , por exemplo, o cálculo de vértices e subgrafos, além de trazer uma coleção de algorítimos que simplificam a análise.

2.7 Spark e R

Através do pacote SparkR é possível trazer diversas ferramentas do Spark para dentro da liguagem R. O SparkR faz com que o R se comunique com o Spark através de uma *API*, permitindo a realização de operações como filtragem, seleção e agregação de dados de forma distribuída. Também é possível rodar *queries* em SQL através do Spark SQL e realizar o treinamento e aplicação de algorítimos de aprendizado de máquinas pelo Spark MLlib. Porém, é importante resaltar que nem todas as funcionalidades do Spark estão presentes no SparkR, existem limitações de diversas componentes como GraphX, Streaming e MLlib. Por exemplo, considerando a documentação do Spark 3.1.2. esta é a lista de modelos de aprendizado de máquina e outras funcionalidades do Spark MLlib que o SparkR tem acesso.

- Classificação
  - Regressão Logística;
  - Perceptron Multicamadas (MLP - Multilayer perceptron);
  - Naive Bayes;
  - Máquina de vetores de suporte (Support vector machine);
  - Máquinas de Fatoração para classificação (Factorization Machines classifier).

- Regressão
  - Modelos de Tempo de Falha Acelerado (AFT - Accelerated failure time);
  - Modelos Lineares Generalizados;
  - Regressão Isotônica;
  - Regressão Linear;
  - Máquinas de Fatoração para regressão. (Factorization Machines regressor).
- Árvores
  - Ávore de Decisão;
  - Gradient Boosting;
  - Florestas Aleatórias.
- Clusterização
  - Bisecting k-means;
  - Modelo de Mistura de Gaussianas (GMM - Gaussian mixture model);
  - K-Means;
  - Alocação Latente de Dirichlet (LDA - Latent Dirichlet Allocation);
  - Power Iteration Clustering (PIC).
- Estatística
  - Teste de Kolmogorov-Smirnov

2.7.1 Funcionamento

A arquitetura que permite que seja possível que programas em R se comuniquem com o Spark funciona da seguinte forma. Primeiro, o programa escrito em R é interpetado pela linguagem e passado para a *JVM*, onde é distribuída entre vários *workers* que realizam as tarefas que lhe foram atribuídas. Após isto, o resultado de cada *worker* é agregado e retorna para o usuário. A figura 9 expõe como parte do processo ocorre.

O objeto dos dados gerado ao apartir do SparkR serão similares aos data frames nativos do R, com a diferença de serem mais eficientes para um grande volumes de dados.

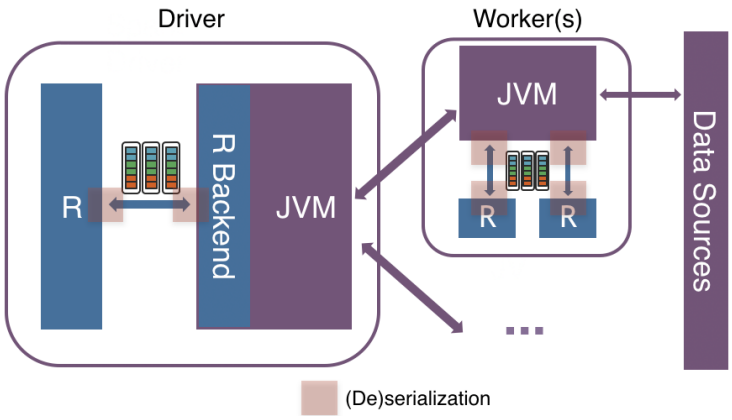

Figura 9: Fluxo R e Spark

2.7.2 Sparklyr

O SparkR não é o único pacote com este intuito, pode-se citar também o sparklyr que além de permitir a utilização do Spark em um ambiente R, também fornece suporte a utilização do pacote dplyr, algo que não é possível no SparkR. Apesar disto, esta monografia focará no SparkR, já que este é mantido pelo mesmo projeto do Spark e é parte do Spark Core. Desta forma, garante-se de que todas as funcionalidades presentes no SparkR possam ser utilizadas, sem a necessidade de atribuir mais um nível de abstração.

2.8 Instalação e Utilização do SparkR

2.8.1 Pré-requisitos

Antes da instalação e utilização do Spark e SparkR, é necessário que alguns requisitos sejam satisfeitos. Primeiramente, deve-se baixar o R no seguinte endereço <https://www.r-project.org/> e instalá-lo seguindo as instruções encontradas no site. Todos os passos descritos nesta monografia levam em consideração a versão 4.1.0 (*Camp Pontanezen*). Para checar a versão do R já instalada, usa-se o comando **version** no console da linguagem. Exemplo de saída:

```
               _
platform       x86_64-w64-mingw32
arch           x86_64
os             mingw32
system         x86_64, mingw32
status
major          4
```

```
minor         1.0
year          2021
month         05
day           18
svn rev       80317
language      R
version.string R version 4.1.0 (2021-05-18)
nickname      Camp Pontanezen
```

Além disso, o Spark foi desenvolvido em Scala, uma linguagem baseada em JVM. Assim, é necessário que se instale a linguagem Java através do endereço <https://www.java.com/pt-BR/download/>.
Para verificar a instalação do Java pode-se usar os comandos:
No terminal do sistema operacional.

Terminal

```
1 java -version
```

```
java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
```

Na linguagem R.

R

```
1 system("java -version")
```

O retorno deve ser similar ao texto abaixo:

```
java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
[1] 0
```

2.8.2 Instalação

Com os pré-requisistos satisfeitos, o próximo passo será a instalação e configuração do Spark. O endereço <https://spark.apache.org/downloads.html> contém algumas opções de versões do Spark para baixar, como mostra na figura abaixo.

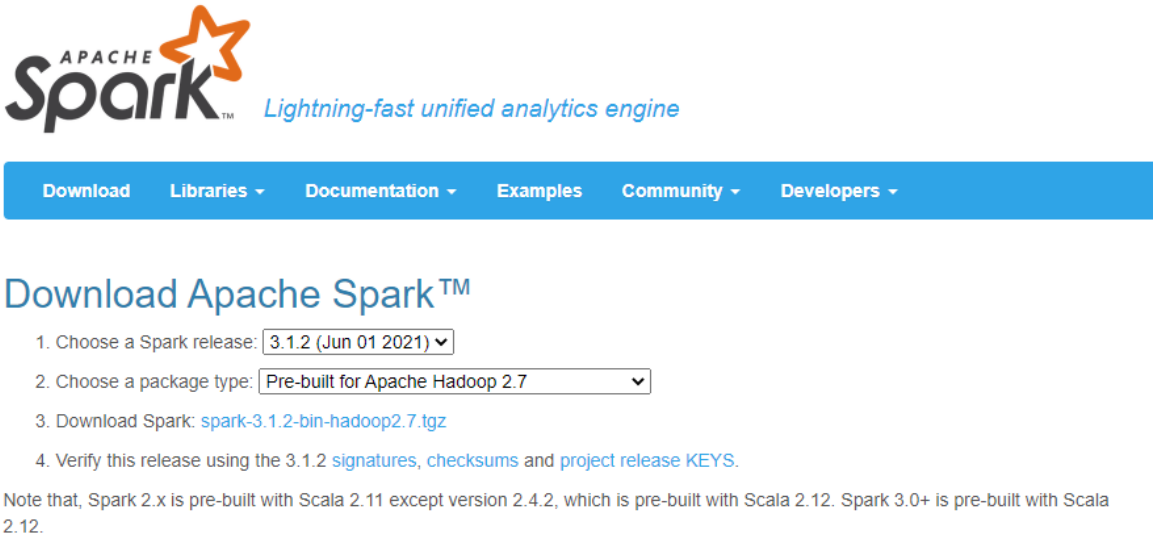

Figura 10: Página de download do Spark

Após selecionar as versões desejadas, deve-se prosseguir com o *download* clicando em **spark-x.x.x-bin-hadoopy.y.tgz**, onde **x** serão os números referentes a versão do Spark e **y** os números referentes a versão do Hadoop. A versão do Spark utilizada nesta monografia será a 3.1.2 e quanto ao Hadoop será utilizada a versão 2.7. O arquivo baixado está compactado, e para realizar sua instalação é necessário realizar a extração de seu conteúdo para uma pasta destino. É importante que se mantenha a integridade desta pasta e de seu conteúdo, bem como sua localização. Nesta monografia o arquivo **spark-3.1.2-bin-hadoop2.7.tgz** foi extraído e movido para a pasta /**home**/**spark**, em um sistema Linux.

2.8.3 Iniciando Uma Sessão Spark

Com os requisitos instalados e o arquivo descompactado, o Spark já esta pronto para ser utilizado conjuntamente com o R. Para isso, é necessário iniciar uma sessão. O código abaixo exemplifica como configurar e iniciar uma sessão do Spark localmente:

starting_spark_session.R

```
# Local de instalação do Spark.
SPARK_HOME = "/home/spark"

# Cria uma variável de ambiente com informação do local de instalação do Spark.
Sys.setenv(SPARK_HOME=SPARK_HOME)

# Carrega o SparkR.
library(SparkR,
       lib.loc=c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))

# Inicia um cluster Local utilizando 3 cores e 2Gb de memória.
sparkR.session(master = "local[3]",
              sparkConfig = list(spark.driver.memory = "2g"))

# Checa a versão do Spark que esta iniciada.
sparkR.version()
```

```
> sparkR.version()
[1] "3.1.2"
```

O Spark possui uma interface gráfica que pode ser acessada após o início de uma sessão. Para acessá-la, considerando que a sessão foi iniciada localmente, basta visitar o endereço local na porta 4040 (porta padrão do Spark) digitando **localhost:4040** em um navegador. Caso o endereço não esteja acessível, a sessão pode não ter sido iniciada corretamente ou a porta 4040 já esta ocupada, neste caso, o Spark irá verificar as portas subsequentes (4041, 4042, ...) até encontrar uma que esteja disponível.

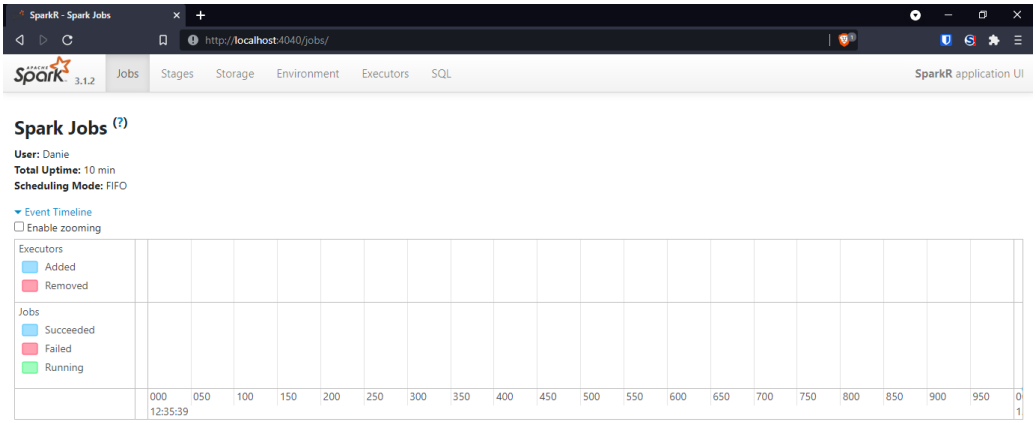

Figura 11: Interface do Spark acessada a partir de um navegador.

2.8.4 Primeiros Passos

Após a início da sessão, pode-se utilizar as diversas ferramentas presentes no SparkR. Esta seção apresentará operações básicas em um pequeno conjunto de dados e exibir algumas diferenças entre os objetos do tipo "data.frame"e "SparkDataFrame".

Suponha-se que existe a necessidade de utilizar o Spark para lidar com dados presentes no R sobre funcionários de uma empresa fictícia. O código abaixo apresenta a função **createDataFrame** que transforma uma lista ou um data.frame do R em um objeto do Spark.

first_steps.R

```
13 # Criar um pequeno conjunto de dados.
14 funcionarios = data.frame(
15   id = c(3, 4, 1, 5, 2),
16   nome = c("Luiz", "Lyncoln", "Gabriel", "Rodolfo", "Leticia"),
17   salario = c(1175.70, 2023.6126, 3116.4971, 1490.11, 2252.9465),
18   data_de_contratacao = as.Date(c("2013-10-11", "2012-07-20", "2018-07-05",
         "2017-08-13", "2018-07-05"))
19 )
20 # Criar um DataFrame do Spark, a partir de um objeto do tipo data.frame do R.
21 funcionarios_spark = createDataFrame(funcionarios)
22
23 # Mostrar no console a difereça entre os conjuntos de dado.
24 funcionarios
25 funcionarios_spark
```

```
> funcionarios
  id    nome  salario data_de_contratacao
1  3    Luiz 1175.700        2013-10-11
2  4 Lyncoln 2023.613        2012-07-20
3  1 Gabriel 3116.497        2018-07-05
4  5 Rodolfo 1490.110        2017-08-13
5  2 Leticia 2252.947        2018-07-05
> funcionarios_spark
SparkDataFrame[id:double, nome:string, salario:double, data_de_contratacao:date]
```

Pode-se observar que no objeto **funcionarios**, os valores do data.frame são mostrados na tela imediatamente, o que não acontece com o objeto **funcionarios_spark**. A seguir, utilizou-se a função **class** para corroborar que os dois objetos realmente possuem estruturas diferentes.

first_steps.R

```
27 # Verificando diferenças das classes
28 class(funcionarios)
29 class(funcionarios_spark)
```

```
> class(funcionarios)
[1] "data.frame"
> class(funcionarios_spark)
[1] "SparkDataFrame"
attr(,"package")
[1] "SparkR"
```

A função **head**, do SparkR e é utilizada para mostrar as n primeiras linhas de um DataFrame, sendo $n = 6$ como padrão.

first_steps.R

```
31 # Obtendo as duas primeiras linhas de um SparkDataFrame
32 head(funcionarios_spark, n = 2)
```

```
> head(funcionarios_spark, n = 2)
  id    nome  salario data_de_contratacao
1  3    Luiz 1175.700        2013-10-11
2  4 Lyncoln 2023.613        2012-07-20
```

A função **collect** a seguir, pertence ao pacote SparkR e é utilizada para mostrar todos os elementos de um DataFrame. Nota-se que neste caso é possível utiliza-lo devido a baixa dimensão dos dados. Além disso, a função **collect** pode ser aplicada para realizar a conversão de um *SparkDataFrame* para um objeto do tipo *data.frame*.

first_steps.R

```
34 # Obtendo todos os dados de um SparkDataFrame
35 collect(funcionarios_spark)
```

```
> collect(funcionarios_spark)
  id    nome  salario data_de_contratacao
1  3    Luiz 1175.700        2013-10-11
2  4 Lyncoln 2023.613        2012-07-20
3  1 Gabriel 3116.497        2018-07-05
4  5 Rodolfo 1490.110        2017-08-13
5  2 Leticia 2252.947        2018-07-05
```

Após executar uma ação no Spark é possível verificar o seu progresso em sua interface. Como executou-se a função **collect**, na interface do Spark aparecerá as estatísticas de sua execução. Como mostrado na figura abaixo:

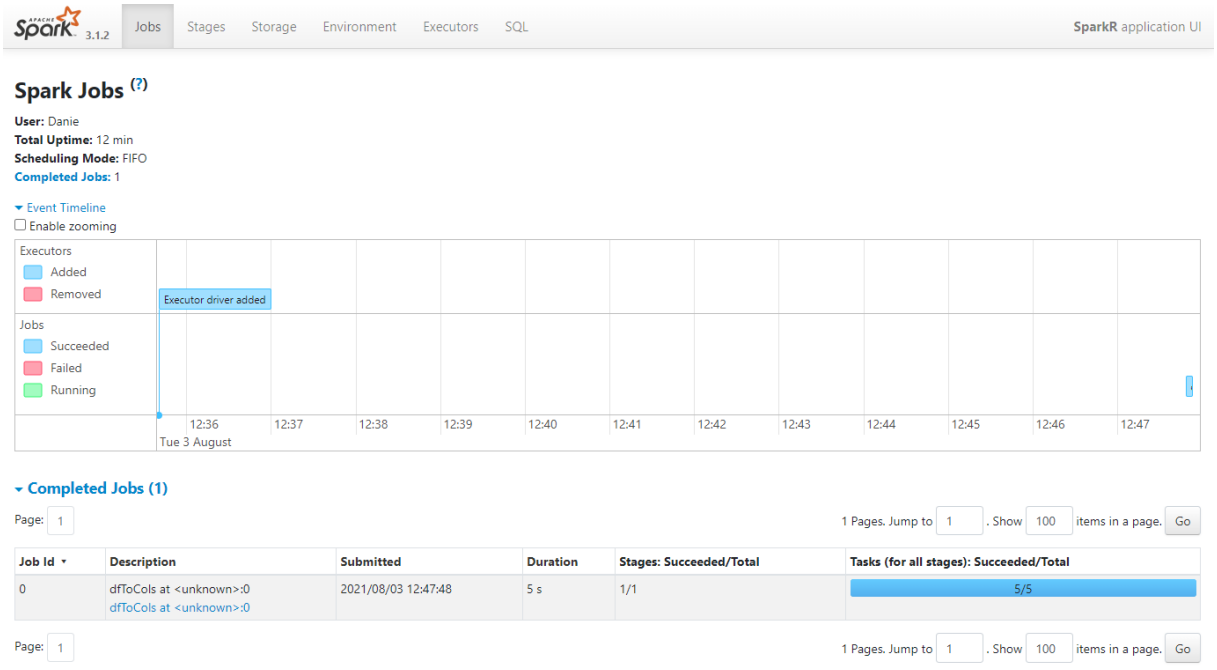

Figura 12: Interface do Spark ao executar uma tarefa

A próxima seção detalha como utilizar funções elementares no SparkR para manipular dados.

2.9 Manipulação de dados com SparkR

As funções do SparkR responsáveis por relizar operações básicas em bancos de dados, em alguns casos, possuem sintaxe similar as encontradas nativamente no R ou em bibliotecas amplamente utilizadas, como por exemplo, o *dplyr*. Esta seção abordará exemplos de utilização destas funções nos dados mostrados na seção anterior.

2.9.1 Operações básicas

Para verificar como um conjunto de dados esta organizado, como nome das colunas e seus respectivos tipos, basta utilizar a função **schema**.

basic_operations.R

```
22 # Obtendo o schema dos dados
23 schema(funcionarios_spark)
```

```
> schema(funcionarios_spark)
StructType
|-name = "id", type = "DoubleType", nullable = TRUE
|-name = "nome", type = "StringType", nullable = TRUE
|-name = "salario", type = "DoubleType", nullable = TRUE
|-name = "data_de_contratacao", type = "DateType", nullable = TRUE
```

2.9.2 Seleção

Assim como no dplyr, se o objetivo é selecionar colunas específicas, pode-se utilizar a função **select**.

selection.R

```
22 # Selecionado apenas a coluna nome
23 collect(
24   select(funcionarios_spark, "nome")
25 )
```

```
> collect(
+   select(funcionarios_spark, "nome")
+ )
     nome
```

```
1   Luiz
2 Lyncoln
3 Gabriel
4 Rodolfo
5 Leticia
```

Para selecionar múltiplas colunas, deve-se adicionar novas *strings* com o nome das colunas separadas por vírgula, por virgula.

selection.R

```
27 # Selecionando duas ou mais colunas
28 collect(
29   select(funcionarios_spark, "nome", "salario")
30 )
```

```
> collect(
+   select(funcionarios_spark, "nome", "salario")
+ )
     nome  salario
1   Luiz 1175.700
2 Lyncoln 2023.613
3 Gabriel 3116.497
4 Rodolfo 1490.110
5 Leticia 2252.947
```

2.9.3 Filtragem

Para a realização da operação de filtragem, basta utilizar a função **filter**, seguido do banco de dados e uma *string* com a lógica de filtragem. Como no exemplo:

filtering.R

```
22 # Filtrando os funcionários que recebem até 2000
23 collect(
24   filter(funcionarios_spark, "salario <= 2000")
25 )
```

```
> collect(
+   filter(funcionarios_spark, "salario <= 2000")
+ )
```

```
  id    nome salario data_de_contratacao
1  3    Luiz 1175.70        2013-10-11
2  5 Rodolfo 1490.11        2017-08-13
```

2.9.4 Criação de novas colunas

Se houver a necessidade de se criar uma nova coluna na qual é uma função de uma das colunas já presentes, deve-se utilizar a função **withColumn**. Ela recebe como argumentos, um *sparkDataFrame*, uma string com o nome da nova coluna e os valores que a nova coluna irá receber. Ao escolher um nome já pertencente a alguma coluna, a mesma será substituída. A seguir, um exemplo no qual deseja-se criar uma variável indicadora para os funcionários que foram contratados antes do ano de 2015. Primeiro, será criada a transformação que se deseja aplicar para gerar a coluna. Pode-se observar que o resultado deste objeto é, na verdade, uma *query* SQL.

creating_new_columns.R

```
22 # Criando condição para coluna identificadora de funcionarios contratados antes
       do ano de 2015
23 condicao = ifelse(funcionarios_spark$data_de_contratacao > "2015-01-01", 0, 1)
24 condicao
```

```
> condicao = ifelse(funcionarios_spark\$data_de_contratacao > "2015-01-01", 0,
    1)
> condicao
Column CASE WHEN (data_de_contratacao > 2015-01-01) THEN 0.0 ELSE 1.0 END
```

Criada a transformação desejada, basta passa-la como um argumento para a função **withColumn**.

creating_new_columns.R

```
26 # Criar uma coluna nomeada de "veterano", indicadora de contratação anterior ao
       ano de 2015.
27 funcionarios_spark = withColumn(funcionarios_spark, "veterano", condicao)
28 collect(funcionarios_spark)
```

```
> funcionarios_spark = withColumn(funcionarios_spark, "veterano", condicao)
> collect(funcionarios_spark)
```

```
  id    nome  salario data_de_contratacao veterano
1  3    Luiz 1175.700        2013-10-11       1
2  4 Lyncoln 2023.613        2012-07-20       1
3  1 Gabriel 3116.497        2018-07-05       0
4  5 Rodolfo 1490.110        2017-08-13       0
5  2 Leticia 2252.947        2018-07-05       0
```

2.9.5 Utilizando SQL no SparkR

O SparkR também pode receber queries de SQL ao invés das funções como, **select**, **filter**, etc. Para isto, é necessário criar uma visualização temporária, isto fará com que o Spark "crie"uma tabela com o nome desejado, assim, as operações serão realizadas em uma tabela com o nome escolhido.

using_sql_in_sparkr.R

```
22 # Criar uma visualização temporária.
23 createOrReplaceTempView(funcionarios_spark, "funcionarios")
```

Criada a visualização, pode-se escrever a *query* em formato de *string* como um argumento da função **sql**. O resultado será um **SparkDataFrame** com os dados da consulta.

using_sql_in_sparkr.R

```
25 # Função que realiza a execução do código SQL.
26 query = sql(
27   "
28   SELECT *,
29   CASE
30     WHEN salario > 2000 THEN 1
31     ELSE 0
32   END AS salario_cat
33   FROM funcionarios
34   "
35 )
36
37 # Transformando SparkDataFrame em data.frame
38 collect(query)
```

```
> query = sql(
```

```
+   "
+   SELECT *,
+   CASE
+     WHEN salario > 2000 THEN 1
+     ELSE 0
+   END AS salario_cat
+   FROM funcionarios
+   "
+ )
>
> # Transformando SparkDataFrame em data.frame
> collect(query)
  id    nome  salario data_de_contratacao salario_cat
1  3    Luiz 1175.700        2013-10-11         0
2  4 Lyncoln 2023.613        2012-07-20         1
3  1 Gabriel 3116.497        2018-07-05         1
4  5 Rodolfo 1490.110        2017-08-13         0
5  2 Leticia 2252.947        2018-07-05         1
```

Nota-se que o mesmo resultado poderia ser obtido utilizando-se das funções nativas do SparkR, como no código abaixo.

```
40 funcionarios_spark = withColumn(funcionarios_spark,
41                                 "salario_cat",
42                                 ifelse(funcionarios_spark$salario > 2000, 1, 0))
43
44 collect(funcionarios_spark)
```

```
> funcionarios_spark = withColumn(funcionarios_spark,
+                                 "salario_cat",
+                                 ifelse(funcionarios_spark$salario > 2000, 1, 0))
>
> collect(funcionarios_spark)
  id    nome  salario data_de_contratacao salario_cat
1  3    Luiz 1175.700        2013-10-11         0
2  4 Lyncoln 2023.613        2012-07-20         1
3  1 Gabriel 3116.497        2018-07-05         1
```

```
4  5 Rodolfo 1490.110       2017-08-13         0
5  2 Leticia 2252.947       2018-07-05         1
```

2.10 Análise Exploratória

A análise exploratória dos dados é a etapa na qual o pesquisador busca calcular estatísticas e criar visualizações que resumem e facilitam o entendimento do objeto de pesquisa. Nesta seção serão mostrados exemplos de como realizar algumas tarefas relacionadas a análise exploratória no contexto de *big data* utilizando o SparkR.

Para esta seção utilizará-se o banco de dados *particles.csv*, que contém dados simulados de velocidade (*speed*) e tamanho (*size*) de três diferentes tipos de partícula (*type*). Os dados podem ser obtido no seguinte enderenço, <https://github.com/Daniel-EST/spark-tcc/blob/master/source/data/particles.csv>.

Para ler um arquivo do tipo *csv* o SparkR basta utilizar a função **read.df**, informando o caminho do arquivo para o argumento *path*, o formato em *source* e o delimitador, vírgula ou ponto e vírgula, em *delimiter*.

exploratory_analysis.R

```
13 # Leitura do arquivo CSV dos dados
14 data = read.df(
15   path = "../../../data/particles.csv",
16   source = "csv",
17   delimiter = ",",
18   inferSchema = "true",
19   header = TRUE
20 )
21
22 head(data)
```

```
> data = read.df(
+   path = "../../../data/particles.csv",
+   source = "csv",
+   delimiter = ",",
+   inferSchema = "true",
+   header = TRUE
+ )
```

```
>
> head(data)
  speed size type
1  6.63 6.77   A
2  6.63 4.84   A
3  6.34 7.33   A
4  6.46 6.03   A
5  7.01 7.53   A
6  7.29 5.23   A
```

2.10.1 Estatística Descritiva

O cálculo de medidas de tendência central, bem como estatísticas de variabilidade, são essenciais para o resumo dos dados, confecção de alguns gráficos e a realização de testes estatísticos. A seguir serão apresentadas ferramentas para obter tais informações através do SparkR.

As funções, **avg**, **percentile_aprox**, **var** e **sd** são exemplos de *column_ aggregate_ functions*, ou seja, funções que agregam resultados de uma coluna e são utilizadas para obter os valores da média, percentis, variância e desvio padrão, respectivamente. Porém, é importante ressaltar que as *column_ aggregate_functions* não retornam o resultado da agregação e sim uma outra coluna, como no exemplo:

Utilizando o objeto *data* criado a partir da leitura do arquivo *exemplo.csv* deseja-se calcular a média da coluna *speed*, referente a velocidade das partículas.

descriptive_statistics.R

```
24 # Verificando tipo de objeto
25 avg(data$speed)
26 class(avg(data$speed))
```

```
> avg(data$speed)
Column avg(speed)
>
> class(avg(data$speed))
[1] "Column"
attr(,"package")
[1] "SparkR"
```

Observa-se que após checar a classe do objeto, utilizando-se da função **class**, o R retorna o tipo *Column* do pacote *SparkR*. Assim, Para que seja possível obter o resultado da agregação é preciso fazer uma seleção, da seguinte forma:

descriptive_statistics.R

```
28 collect(
29   select(data, avg(data$speed))
30 )
```

```
> collect(
+   select(data, avg(data$speed))
+ )
  avg(speed)
1   7.684628
```

A coluna *speed* apresentou uma média de aproximadamente 7,68. Para calcular as demais medidas, pode-se adicionar as funções mostradas anteriormente com suas devidas colunas a serem agregadas como os próximos argumentos da função **select**.

Na função **percentile_approx** deve-se passar o argumento *percentage*, um valor de 0 (zero) até 1 (um) que indicará o percentil desejado, neste caso, selecionou-se 0.5 para que se fosse obtida a mediana.

descriptive_statistics.R

```
33 # Obtendo: média, variância e mediana das velocidades
34 collect(
35   select(data,
36          avg(data$speed),
37          var(data$speed),
38          percentile_approx(data$speed, percentage=.5))
39   )
```

```
> collect(
+   select(data,
+          avg(data$speed),
+          var(data$speed),
+          percentile_approx(data$speed, percentage=.5))
+   )
  avg(speed) var_samp(speed) percentile_approx(speed, 0.5, 10000)
```

```
1   7.684628        1.06148                                  7.67
```

Através deste código, observa-se que a coluna *speed* apresentou variância próxima a 1,06 e mediana[2] 7,67.

Outra forma de mostrar dados agregados é utilizando a função **agg** que retorna um *SparkDataFrame* já com os valores agregados. Além disto, esta função é funciona com diversas formas de sintaxe, são alguns exemplos:

- Passando o nome da coluna como argumento seguida da *string* com a operação de agregação deseada;

descriptive_statistics.R

```
41 # Obtendo a média da velocidade usando a função agg
42 collect(
43   agg(data, speed = "avg") # Coluna = "avg, variance, etc"
44 )
```

```
> collect(
+   agg(data, speed = "avg") # Coluna = "avg, variance, etc"
+ )
  avg(speed)
1   7.684628
```

- Passando o nome de uma nova coluna como argumento, seguida da função de agregação da operação desejada;

descriptive_statistics.R

```
46 # Nome da nova coluna como argumento seguida da operação desejada
47 collect(
48   agg(data, media_da_velocidade = avg(data$speed))
49 )
```

```
> collect(
+   agg(data, media_da_velocidade = avg(data$speed))
+ )
  media_da_velocidade
```

[2]A função **percentile_approx** calcula a percentil aproximado, baseando-se no consumo de memória. Esta acurácia pode ser controlada através do argumento *accuracy*, possuindo um valor padrão de 1000. O erro relativo da aproximação pode ser calculado como $\frac{1}{accuracy}$

```
1          7.684628
```

Agregação por grupos

A função **agg** em conjunto com a função **groupBy** é capaz de realizar todas as operações mostradas anteriormente, mas com uma separação por grupos. Deseja-se calcular as médias do tamanho e velocidade para cada tipo de partícula do banco de dados *particles.csv* presente no objeto *data*.

descriptive_statistics.R

```
55 collect(
56   agg(groupBy(data, "type"), # Agrupamento pela coluna "type"
57       media_tamanho = avg(data$size),
58       media_velocidade = avg(data$speed))
59 )
```

```
> collect(
+   agg(groupBy(data, "type"), # Agrupamento pela coluna "type"
+       media_tamanho = avg(data$size),
+       media_velocidade = avg(data$speed))
+ )
  type media_tamanho media_velocidade
1    B     5.128333       7.596216
2    C     5.570392       8.360800
3    A     6.483824       6.786471
```

Caso haja a necessidade do pesquisador realizar o agrupamento por diversas colunas, o mesmo deve adicionar novos argumentos do tipo *strings*, com o nome das colunas desejadas, separando-as por vírgula.

Por fim, caso o usuário possua maior familiaridade com SQL, todas estas formas de reportar resultados podem ser substituídas por *queries* utilizando a função **sql**, como visto na seção Utilizando SQL no SparkR.

2.10.2 Visualização de Dados

Com o intuito de analisar e apresentar resultados é comum que o pesquisador utilize elementos visuais que resumem informações que podem ser relevantes para o estudo. Os gráficos possuem grande importância nos métodos tradicionais e não é diferente no

universo do *big data*. Através do SparkR, consegue-se obter os insumos necessários para a confecção de diversos gráficos. A seguir serão mostrados alguns exemplos básicos de construção de gráficos.

Barras

Os gráficos de barras são utilizados para resumir um conjunto de dados categóricos, através de contagens, frequências relativas, médias, etc. No exemplo a seguir, constrói-se um gráfico de barras para avaliar a frequência absoluta de partículas para cada tipo.

data_visualization.R

```
24 # Obtendo valores para realização para gráfico de barras
25 barplot_values = collect(count(groupBy(data, "type")))
26 barplot_values
```

```
> barplot_values = collect(count(groupBy(data, "type")))
> barplot_values
  type count
1    B    37
2    C    51
3    A    34
```

Neste exemplo, a função de **groupBy** é utilizada para agrupar os dados por tipo e em seguida realiza-se a contagem de linhas através da função **count**. Por fim, a função **collect** faz com que o Spark processe e retorne ao computador os resultados. Com o *data.frame* com as informações desejadas, basta utilizar a ferramenta de preferência para gerar o gráfico. Neste exemplo foi utilizado o pacote *ggplot*.

data_visualization.R

```
28 require(ggplot2) # Importar pacote para criação de gráficos
29
30 ggplot(barplot_values, aes(x = type, y = count)) +
31   geom_bar(stat = "identity", col = "black") +
32   ggtitle("Quantidade de partículas por tipo") +
33   xlab("Tipo") + ylab("Frequência")
```

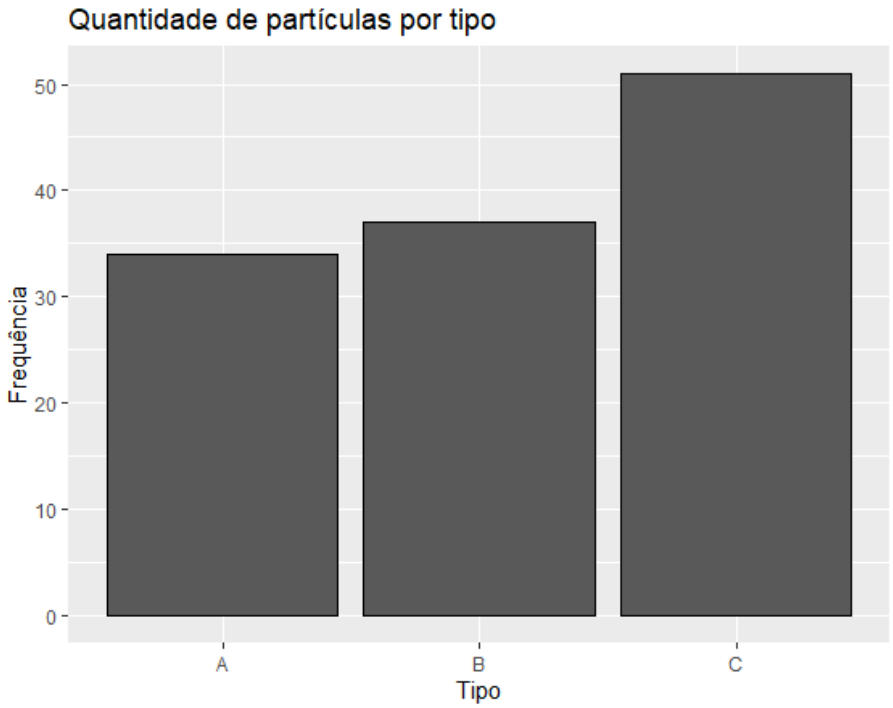

Figura 13: Exemplo de gráfico de barras

Histogramas

Apesar de serem visualmente semelhantes aos gráficos de barras, os histogramas são utilizados para realizar contagens em variáveis contínuas. Para construir este tipo de gráfico, o pesquisador deve fixar o número de intervalos de igual amplitude, disjuntos e abertos a esquerda, que a visualização terá, este número é comumente chamado por bins. Apesar de não existir um número ideal de *bins*, são comumente utilizadas a fórmula de Sturges ou algum tipo de função que leve em consideração a amplitude dos dados. Após a definição dos intervalos, é realizada a contagem dos valores que pertencem a cada um dos conjuntos.

O SparkR possui a função **histogram**, que realiza todos os passos para a realização de histograma e de forma eficente. Esta função possui os argumentos *df* que recebe um objeto do tipo *SparkDataFrame*, *col* uma *string* ou um objeto do tipo *Column* na qual deseja-se obter o histograma e por fim o argumento *nbins* onde define o número de *bins* e possui valor padrão 10 (dez).

No exemplo a seguir, constrói-se um histograma para avaliar a distribuição do tamanho das partículas.

data_visualization.R

```
35 # Obtendo valores para realização de um histograma
```

```
36 hist_values = histogram(data, "size", nbins = 10)
37 hist_values
```

```
> hist_values = histogram(data, "size", nbins = 10)
> hist_values
   bins counts centroids
1     0      4     3.307
2     1     10     3.921
3     2     19     4.535
4     3     20     5.149
5     4     21     5.763
6     5     21     6.377
7     6     16     6.991
8     7      6     7.605
9     8      2     8.219
10    9      2     8.833
```

Os *centroids* são os pontos médios dos intervalos e são necessários para a confecção no *ggplot*.

data_visualization.R

```
39 ggplot(hist_values, aes(x = centroids, y = counts)) +
40   geom_bar(stat = "identity", col = "black") +
41   ggtitle("Histograma da velocidade das partículas") +
42   xlab("Velocidade") + ylab("Frequência")
```

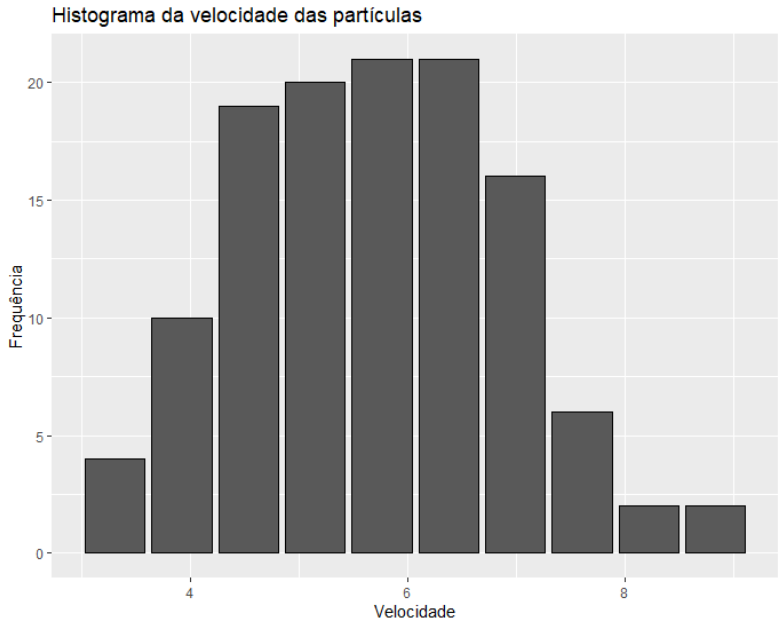

Figura 14: Exemplo de Histograma

Boxplots

Os diagramas de caixas ou boxplots, como são mais conhecidos, são gráficos que auxiliam na observação da dispersão dos dados, utilizando como base no quantis observados. As caixas possuem 3 linhas horizontais referentes ao primeiro quantil, a mediana e o terceiro quantil, respectivamente, além de possuírem duas linhas verticais, conhecidas como *whiskers* ou bigodes, que representam o mínimo e máximo das observações sem a presença de valores discrepantes. Estes bigodes são calculados geralmente da seguinte forma:

Para os valores de mínimo:

$$LI = Q_1 - 1,5(Q_3 - Q_1)$$

Para os valores de máximo:

$$LS = Q_3 + 1,5(Q_3 - Q_1)$$

Onde a diferença $Q_3 - Q_1$ é conhecida como, intervalo inter-quantílico.

data_visualization.R

```
44 # Obtendo valores para realização para o boxplot
45 boxplot_values = collect(
46   agg(groupBy(data, "type"),
47       med = percentile_approx(data$speed, percentage=.5),
```

```
48      q1 = percentile_approx(data$speed, percentage=.25),
49      q3 = percentile_approx(data$speed, percentage=.75))
50 )
51 boxplot_values
```

```
> boxplot_values = collect(
+   agg(groupBy(data, "type"),
+       med = percentile_approx(data$speed, percentage=.5),
+       q1 = percentile_approx(data$speed, percentage=.25),
+       q3 = percentile_approx(data$speed, percentage=.75))
+ )
> boxplot_values
  type  med   q1   q3
1    B 7.64 7.25 8.18
2    C 8.35 7.94 8.81
3    A 6.87 6.36 7.16
```

Para o obter os valores dos quantis para cada tipo de partícula, utilizou-se a função **agg** em conjunto da função **groupBy** responsável pelo agrupamento dos tipos e a função **percentile_approx** que retorna o valor dos percentis aproximados.

data_visualization.R

```
53 ggplot(boxplot_values,
54       aes(x = type,
55           ymin = q1 - 1.5 * (q3 - q1),
56           lower = q1,
57           middle = med,
58           upper = q3,
59           ymax = q3 + 1.5 * (q3 - q1))) +
60   geom_boxplot(stat = "identity") +
61   ggtitle("Boxplot da velocidade das partículas por tipo de partícula") +
62   xlab("Tipo") + ylab("Velocidade")
```

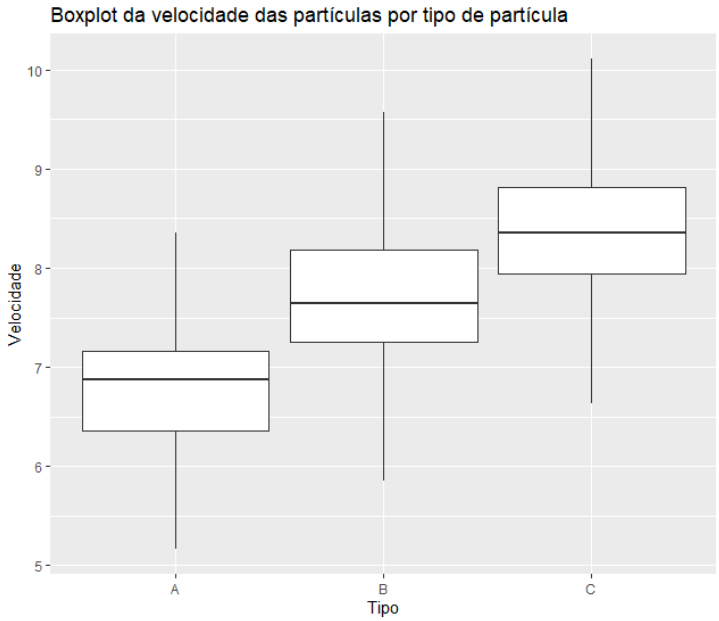

Figura 15: Exemplo de box-plot

Outros

Devido ao grande volume de dados, em alguns casos buscam-se alternativas as visualizações mais tradicionais. Um exemplo disto é o gráfico de dispersão, em *big data* torna-se muito custoso verificar todos os pares de dados, para que assim sejam mostrados no gráfico. Além disso, devido a grande quantidade de pontos, a visualização pode ficar poluída e pouco informativa. Neste caso uma alternativa seria a criação de um histograma bivariado.

Suponha que deseja-se verificar a relação entre a velocidade e o tamanho das partículas, utilizando um gráfico de dispersão obtem-se:

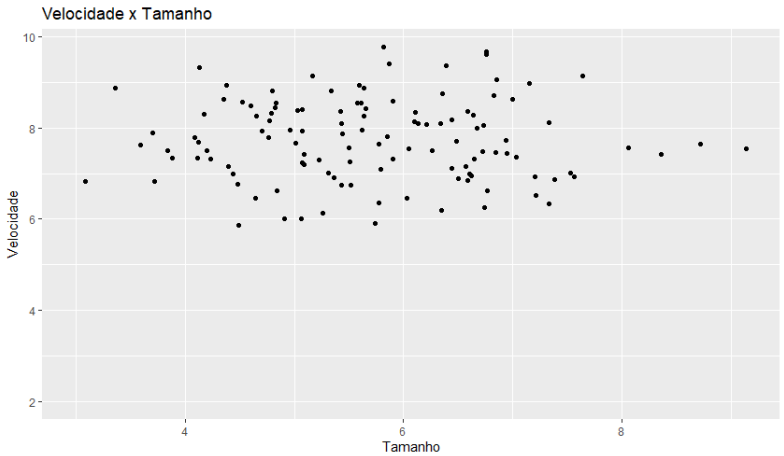

Figura 16: Gráfico de dispersão

O código abaixo mostra como gerar um histograma bivariado para verificar a relação entre as variáveis *speed* e *size*.

data_visualization.R

```
65 # Código para criação de um histograma bivariado
66
67 # Definir a quantidade de bins
68 nbin = 13
69
70 # Calcular mínimo de x
71 x_min = collect(agg(data, min(data$size)))
72 # Calcular máximo de x
73 x_max = collect(agg(data, max(data$size)))
74 # Definir os intervalos para os bins de x
75 x_bin = seq(floor(x_min[[1]]),
76            ceiling(x_max[[1]]),
77            length = nbin)
78
79 # Calcular mínimo de y
80 y_min = collect(agg(data, min(data$speed)))
81 # Calcular máximo de y
82 y_max = collect(agg(data, max(data$speed)))
83 # Definir os intervalos para os bins de y
84 y_bin = seq(floor(y_min[[1]]),
85            ceiling(y_max[[1]]),
86            length = nbin)
87
88 # Calcular tamanho do intervalo dos bins de x e y
89 x_bin_width = x_bin[[2]] - x_bin[[1]]
90 y_bin_width = y_bin[[2]] - y_bin[[1]]
91
92 # Calcular a qual bin pertece cada valor observado
93 graph_data = withColumn(data, "x_bin", ceiling((data$size - x_min[[1]]) /
       x_bin_width))
94 graph_data = withColumn(graph_data, "y_bin", ceiling((data$speed - y_min[[1]])
       / y_bin_width))
95 graph_data = mutate(graph_data, x_bin = ifelse(graph_data$x_bin == 0, 1,
       graph_data$x_bin))
```

```
graph_data = mutate(graph_data, y_bin = ifelse(graph_data$y_bin == 0, 1,
    graph_data$y_bin))

graph_data = collect(agg(groupBy(graph_data, "x_bin", "y_bin"),
                         count = n(graph_data$x_bin)))

ggplot(graph_data, aes(x = x_bin, y = y_bin, fill = count)) +
  geom_tile() +
  scale_fill_distiller(palette = "Blues", direction = 1) +
  ggtitle("Velocidade x Tamanho") +
  xlab("Tamanho") + ylab("Velocidade")
```

Assim, gerou-se um gráfico que acumula os pontos do gráfico de dispersão em quadrantes, facilitando o processamento e a visualização.

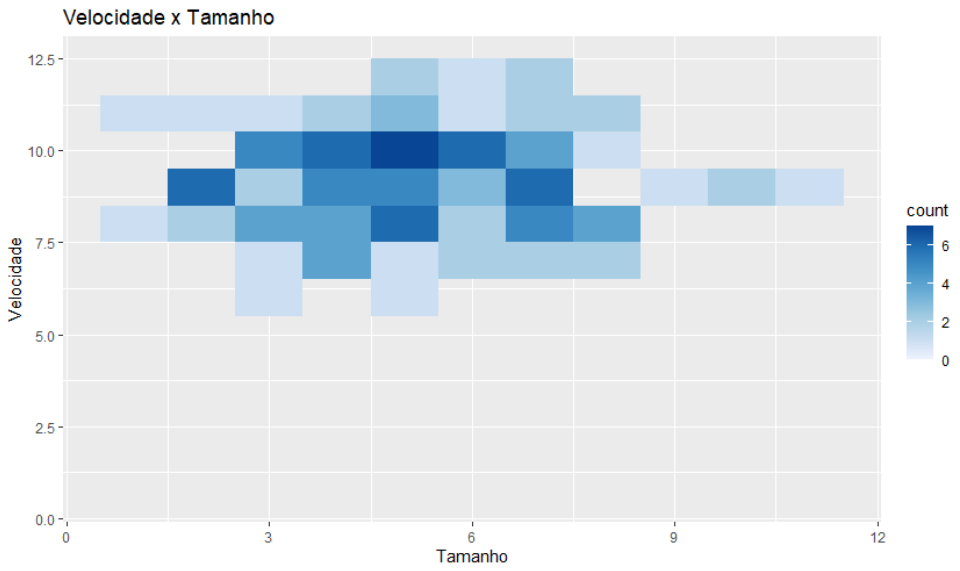

Figura 17: Exemplo histograma bivariado

2.11 Aprendizado de máquina no SparkR

Seja para classificar dados não rotulados ou resolver problemas de regressão, o aprendizado de máquinas é conjunto de boas práticas junto ao treinamento de modelos que visa realizar predições em dados. Como apresentado na seção 2.6.4, o SparkR possui algumas funções para a aplicação de modelos de aprendizado de máquina. Porém, algumas ferramentas como a criação de *pipelines*, alguns algorítimos de treinamento e métricas para avaliação dos modelos não foram implementadas no SparkR até o momento, entretanto podem ser realizados através do Sparklyr.

2.11.1 Preparação do banco de dados

Uma parte importante da modelagem é a preparação dos dados para o treino, um dos problemas facilmente encontrados são a presença de dados faltantes. O código a seguir utiliza-se da função **isNull**, para verificar a presença de dados faltantates nas colunas relacionadas a velocidade e tamanho.

data_preparation.R

```
19 # Verificando quantidade de dados faltantes
20 collect(
21   agg(
22     groupBy(data,
23             isNull(data$speed),
24             isNull(data$size)),
25     count(data$type)
26     )
27   )
```

```
> collect(
+   agg(
+     groupBy(data,
+             isNull(data$speed),
+             isNull(data$size)),
+     count(data$type)
+     )
+   )
  (speed IS NULL) (size IS NULL) count(type)
1            TRUE          FALSE           1
```

```
2          FALSE         FALSE        120
3          FALSE          TRUE          1
```

Após a execução do código, foi detectado um valor faltante na coluna *speed* e um valor faltante na coluna *size*. Para tratar estas lacunas, pode-se recorrer a uma técnica conhecida como imputação. A imputação consiste em atribuir um valor para as características faltantes, existem inúmeras formas de realizar tal tarefa. Neste exemplo optou-se por imputar os dados faltantes utilizando-se da média da coluna. Assim, se se não existe a informação da velocidade para determinada partícula, esta será substituída pela média das velocidades.

A função **fillna** identifica os valores faltantes e os preenche com os valores desejados. Pode-se passar uma lista nomeada para que diferentes valores sejam atribuídos a diferentes colunas, como abaixo:

data_preparation.R

```
29 # Calculando média para imtação dos dados
30 mean_speed = collect(select(data, avg(data$speed)))[[1]]
31 mean_size = collect(select(data, avg(data$size)))[[1]]
32
33 # Imputando dados faltantes respectivos as suas colunas
34 data = fillna(data, list("speed" = mean_speed,
35                          "size" = mean_size))
```

Com os dados faltantes tratados, pode-se separar a base em treino e teste, que serão utilizados posteriormente para treinar o modelo e avaliá-lo simulando a entrada de novos dados, respectivamente. A função **randomSplit** recebe como argumentos o *SparkDataFrame* e um vetor de pesos, esta função retorna uma lista que possui *SparkDataFrames* com uma quantidade de linhas proporcional aos pesos escolhidos. O argumento *seed* fixa a separação dos resultados, para fins de reprodutibilidade.

```
37 df_list = randomSplit(data, c(7,3), seed = 210828)
38 df_list
```

```
> df_list = randomSplit(data, c(7,3), seed = 210828)
> df_list
[[1]]
SparkDataFrame[speed:double, size:double, type:string]
```

```
[[2]]
SparkDataFrame[speed:double, size:double, type:string]
```

Ao final separou-se a lista em dois objetos.

```
41 train = df_list[[1]]
42 test = df_list[[2]]
```

2.11.2 Treinamento o modelo

O SparkR possui uma série de modelos prontos para serem utilizados, como mostrado na seção 2.6.4, o código abaixo exemplifica a utilização de um modelo linear para a previsão da velocidade das partículas através de seus tamanho. A função **spark.lm** aplica este modelo e recebe como parâmetros um *SparkDataFrame* para treinamento e um objeto do tipo *formula* com as variáveis resposta e explicativas, esta sintaxe é similar a função *lm* do pactote *stats* já pertencente ao R.

model_trainning.R

```
45 # Treinammento do modelo
46 model = spark.lm(data = train, size ~ speed)
47 # Verificar os parâmetros calculados do modelo
48 summary(model)
```

```
> model = spark.lm(data = train, size ~ speed)
> # Verificar os parâmetros calculados do modelo
> summary(model)
$coefficients
             Estimate
(Intercept) 6.5899001
speed      -0.1336585

$numFeatures
[1] 1
```

2.11.3 Avaliação do modelo

Após o treinamento do modelo é importante que se possa avaliá-lo, desta forma consegue-se mensurar sua capacidade de predição além de tornar possível a comparação entre diferentes parâmetros e algorítimos de modelagem. Diferentes métricas são utilizadas dependendo do tipo de modelo criado, em problemas de classificação podem ser utilizadas matrizes de confusão, sensibilidade e especificidade. Já em modelos de regressão pode-se utilizar, erro absoluto, erro quadrático médio, R^2 ou coeficiente de regressão, R^2 Ajustado, entre outros.
O código a seguir mostra como avaliar um modelo de regressão utilizando o R^2 e o R^2 ajustado na amostra de teste.

Primeiro serão realizadas as predições do banco de dados de teste.

model_evaluation.R

```
49 # Utilizando o modelo para prever os resultados dentro da amostra treino
50 predictions = predict(model, test)
51 head(predictions)
```

Após as predições, calcula-se a soma dos quadrados totais denotada por SQ_{tot} e a soma dos quadrados dos resíduos denotada por SQ_{res} da seguinte forma.

$$SQ_{tot} = \sum_{i=1}^{n} (y_i - \overline{y})^2$$

$$SQ_{res} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Onde y_i a observação da variável resposta do indivíduo i, $\overline{y}$ a média da variável resposta e $\hat{y}_i$ a predição realizada pelo modelo no indivíduo i.

model_evaluation.R

```
53 # Calculando a média da variável resposta
54 y_avg = collect(agg(predictions, avg(predictions$label)))[[1]]
55
56 # Calculando SQ_res e SQ_tot
57 df =
58   agg(predictions,
59       sq_res = sum((predictions$label - predictions$prediction)^2), # Somando os
           quadrado dos resíduos
```

```
60      sq_tot = sum((predictions$label - y_avg)^2)) # Somando os quadrados totais
61
62 SSR = collect(select(df, df$sq_res))[[1]]
63 SST = collect(select(df, df$sq_tot))[[1]]
```

Após a obtenção das somas dos quadrados calcula-se o R^2 da seguinte forma:

$$R^2 = 1 - \frac{SQ_{res}}{SQ_{tot}}$$

model_evaluation.R

```
65 # Calculando R
66 Rsq = 1 - (SSR/SST)
```

Para o calculo do R^2 Ajustado basta aplicar a seguinte formula com os valores obtidos anteriormente

$$R^2_{ajustado} = 1 - \frac{(1 - R^2)(n - 1)}{n - p}$$

model_evaluation.R

```
68 # Obtendo o nmero de parâmetros do modelo
69 p = summary(model)$numFeatures + 1
70
71 #  Nmero  de observações
72 n = nrow(predictions)
73
74 # Calculando o R Ajustado
75 aRsq = 1 - (((1 - Rsq)*(n - 1))/(n - p))
```

model_evaluation.R

```
77 sprintf(" R : %.5f", Rsq)
78 sprintf("RAjustado: %.5f", aRsq)
```

Neste exemplo os valores do R^2 e do $R^2_{ajustado}$ foram próximos de zero, ou seja, trata-se de um modelo não representativo.

2.11.4 Salvando e carregando o modelo treinado

Após a obtenção do modelo para que não seja necessário treina-lo novamente é indispensável que o salve. As funções **write.ml** e **read.ml**, escrevem e carregam, respectivamente, os modelos nos caminhos desejados.
Salvando o modelo:

saving_loading_model.R

```
80 # Salvando o modelo
81 write.ml(model, "./model/lm_particles")
```

Carregando o modelo para o objeto *model_loaded*

saving_loading_model.R

```
83 # Carregando o modelo
84 model_loaded = read.ml("./model/lm_particles")
```

3 Análise dos Resultados

A partir das funções e ferramentas apresentadas no capítulo 2, Materiais e Métodos, será realizada uma análise e treinamento de um modelo para um conjunto de dados com maiores dimensões, com o intuito de demonstrar o potencial do Spark.

3.1 Apresentação do Banco de Dados

O banco de dados possui diversas informações de carros usados, obtidos através de um *crawler* em um *site* de vendas online dos Estados Unidos da América, possui cerca de 9,7 Gigabytes, 3 milhões de linhas e 66 colunas. A descrição das colunas e informações do banco de dados pode ser verificadas no apêndice 1. Os dados podem ser adquiridos no seguinte endereço, <https://www.kaggle.com/ananaymital/us-used-cars-dataset>. Dito isso, o objetivo será determinar o preço de um carro usado, utilizando-se das características mostradas no anúncio *online*.

O código abaixo carrega os dados, mostra as dimensões e as colunas:

used_cars.R

```
SPARK_HOME = "D:/Spark"

Sys.setenv(SPARK_HOME=SPARK_HOME)


library(ggplot2)
library(SparkR,
       lib.loc=c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))

sparkR.session(master = "local[*]",
               sparkConfig = list(spark.driver.memory = "1g"))

```

```
13 data = read.df(
14   path = "../../../data/used_cars_data.csv",
15   source = "csv",
16   delimiter = ",",
17   inferSchema = "true",
18   na.strings = "None",
19   header = TRUE
20 )
21
22 printSchema(data)
23 nrow(data)
24 ncol(data)
```

3.2 Tratamento dos dados

Os dados obtidos através do *crawler* possuem algumas incongruências. Assim tornou-se extremamente necessário que os dados fossem tratados antes de treinar o modelo. Algumas colunas foram desconsideradas durante este estudo, sendo elas: *vin*, *sp_name*, *sp_id*, *daysonmarket*, *description*, *franchise_dealer*, *franchise_make*, *listed_date*, *listing_id*, *main_picture_url*, *major_options*, *latitude*, *longitude*, *trimId*, *trim_name*, *dealer_zip*, *interior_color*, *exterior_color*, *model_name*, *engine_cylinders*, *transmission_display*, *wheel_system_display*, *savings_amount*, *salvage*, *theft_title*. Estas colunas foram removidas com base em diferentes critérios, quando por exemplo, apenas se tratavam de identificadores do anúncio, geolocalização do vendedor, informações secundárias como o texto aberto que descreve o veículo no *site* bem como sua imagem ou a presença de pares de colunas com informações semelhantes, como as colunas *listing_color* e *exterior_color*.

used_cars.R

```
26 data = drop(data,
27            c("vin",
28              "sp_name",
29              "sp_id",
30              "daysonmarket",
31              "description",
32              "franchise_dealer",
33              "franchise_make",
```

```
34              "listed_date",
35              "listing_id",
36              "main_picture_url",
37              "major_options",
38              "latitude",
39              "longitude",
40              "trimId",
41              "trim_name",
42              "dealer_zip",
43              "interior_color",
44              "exterior_color",
45              "model_name",
46              "engine_cylinders",
47              "transmission_display",
48              "wheel_system_display",
49              "savings_amount",
50              "salvage",
51              "theft_title"))
```

Colunas que apresentavam valores faltantes maiores que 50% da base foram excluídas. Além disso, também foram removidas linhas que apresentavam preços faltantes, carros descritos como anteriores à 1990 e posteriores à 2021, como no código:

used_cars.R

```
54 data = filter(data, isNotNull(data$price))
55 data = filter(data, data$year > 1990 & data$year <= 2021)
56
57
58 cols = lapply(columns(data), \(x) alias(count(data[[x]]), x))
59
60 notnull_count = collect(select(data, cols))
61 n = nrow(data)
62
63 notnull_prop = (notnull_count/n)
64
65 cols = colnames(notnull_prop)[notnull_prop > 0.5]
66 data = select(data, cols)
```

Após a remoção destas colunas, necessitou-se que algumas colunas fossem convertidas para o seus devidos formatos, algumas colunas como *back_ legroom* apresentavam textos com a distância em polegadas. Utilizou-se a função **regexp_ replace** para remover o texto que acompanhava o número e a função **cast** para converter a coluna para seu devido formato. As variáveis *power* e *torque* possuíam na mesma coluna informações sobre RPM (rotações por minuto) e força dada em cavalos de força, portanto foram utilizadas as funções **regexp_ extract** e **regex_ replace** para separar em diferentes colunas.

used_ cars.R

```
68 convert_col = c("back_legroom",
69                 "front_legroom",
70                 "fuel_tank_volume",
71                 "height",
72                 "length",
73                 "maximum_seating",
74                 "wheelbase",
75                 "width")
76
77 for(col in convert_col){
78   data = withColumn(data, col, regexp_replace(data[[col]], " \\w+", ""))
79   data[[col]] = cast(data[[col]], "double")
80 }
81
82 data$price = cast(data$price, "double")
83 data$back_legroom = cast(data$back_legroom, "double")
84 data$city_fuel_economy = cast(data$city_fuel_economy, "double")
85 data$engine_displacement = cast(data$engine_displacement, "double")
86 data$horsepower = cast(data$horsepower, "double")
87 data$mileage = cast(data$mileage, "double")
88 data$seller_rating = cast(data$seller_rating, "double")
89 data$highway_fuel_economy = cast(data$highway_fuel_economy, "double")
90
91 data = withColumn(data, "power_rpm", regexp_extract(data[["power"]],
       "\\d+,\\d+", 0))
92 data = withColumn(data, "power_rpm", regexp_replace(data[["power_rpm"]], ",",
       ""))
93 data$power_rpm = cast(data$power_rpm, "double")
94
```

```
95 data = drop(data, "power")
96
97 data = withColumn(data, "torque_power", regexp_extract(data[["torque"]],
       "\\d+", 0))
98 data$torque_power = cast(data$torque_power, "double")
99 data = withColumn(data, "torque_rpm", regexp_extract(data[["torque"]],
       "\\d+,\\d+", 0))
100 data = withColumn(data, "torque_rpm", regexp_replace(data[["torque_rpm"]], ",",
       ""))
101 data$torque_rpm = cast(data$torque_rpm, "double")
102
103 data = drop(data, "torque")
104
105 data$year = cast(data$year, "integer")
106 data$owner_count = cast(data$owner_count, "integer")
107 data$maximum_seating = cast(data$maximum_seating, "integer")
```

3.3 Análise Descritiva

Após o tratamento das colunas tornou-se possível a realização de análises descritivas, assim tomou-se como primeiro passo calcular as médias das variáveis numéricas.

used_cars.R

```
110 # Seprar variáveis categóricas das numéricas
111 categoric = c()
112 numeric = c()
113 for(col in dtypes(data)){
114   if(col[[2]] != "double"){
115     categoric = c(categoric, col[[1]])
116   } else {
117     numeric = c(numeric, col[[1]])
118   }
119 }
120
121 means = lapply(numeric, \(x) alias(avg(data[[x]]), x))
122 means = collect(select(data, means))
```

Para compreender o comportamento de algumas variáveis numéricas como, *city_fuel_economy*, *highway_fuel_economy* e *mileage* foram gerados histogramas.

used_cars.R

```
127 options(scipen = 999)
128 plot_histogram = function(data, colname, nbins = 32){
129   hist = histogram(data, data[[colname]], nbins = nbins)
130 
131   p = ggplot(hist, aes(x = centroids, y = counts)) +
132     geom_bar(stat = "identity", col = "black") +
133     ggtitle("") +
134     xlab(colname) + ylab("Frequência") +
135     theme_minimal()
136 
137   print(p)
138 
139   return(list(colname, hist))
140 }
141 col_hist = c("city_fuel_economy", "highway_fuel_economy", "mileage")
142 histograms = lapply(col_hist, \(x) plot_histogram(data, x, nbins = 32))
```

Gerou-se o seguinte histograma:

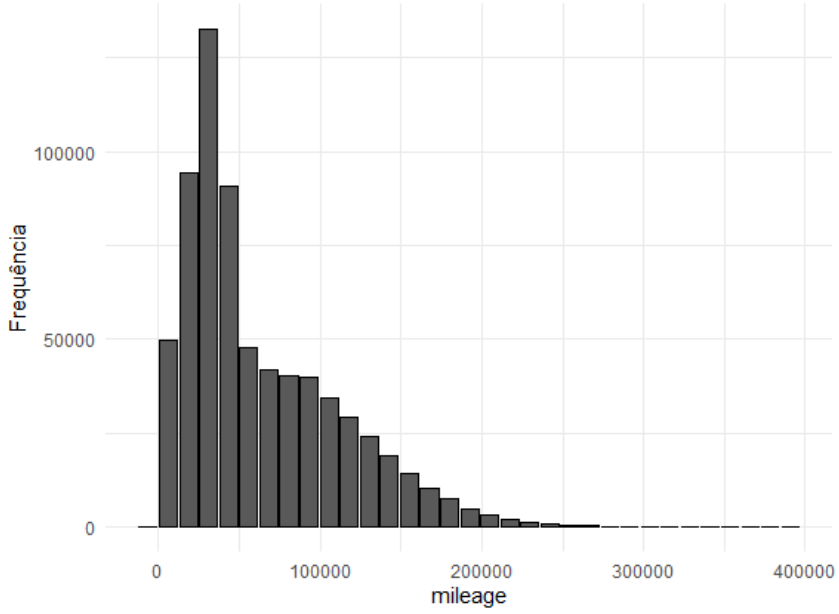

Figura 18: Distribuição das milhas percorridas

Também calculou-se os mínimos e os máximos de diversas colunas, tornando-se possível remover as linhas que apresentavam valores negativos para o preço.

used_cars.R

```
144 collect(
145   agg(data,
146       max_price = max(data$price),
147       min_price = min(data$price),
148       max_maximum_seating = max(data$maximum_seating),
149       min_maximum_seating= min(data$maximum_seating),
150       max_owner_count = max(data$owner_count),
151       min_owner_count = min(data$owner_count),
152       max_seller_rating = max(data$seller_rating),
153       min_seller_rating = min(data$seller_rating))
154 )
155
156 data = transform(data, price = ifelse(data$price <= 0, NA, data$price))
157 data = filter(data, isNotNull(data$price))
```

Com o intuito de verificar outras disparidades com a variável resposta relizou-se um boxplot de seus valores.

used_cars.R

```
159 boxplot_price = collect(
160   agg(data,
161       med = percentile_approx(data$price, percentage=.5),
162       q1 = percentile_approx(data$price, percentage=.25),
163       q3 = percentile_approx(data$price, percentage=.75))
164   )
165
166 ggplot(boxplot_price,
167        aes(x = "",
168            ymin = q1 - 1.5 * (q3 - q1),
169            lower = q1,
170            middle = med,
171            upper = q3,
172            ymax = q3 + 1.5 * (q3 - q1))) +
173   geom_boxplot(stat = "identity") +
174   ggtitle("Boxplot do Preço") +
```

```
175   ylab("Preço") + xlab("") +
176   theme_minimal()
```

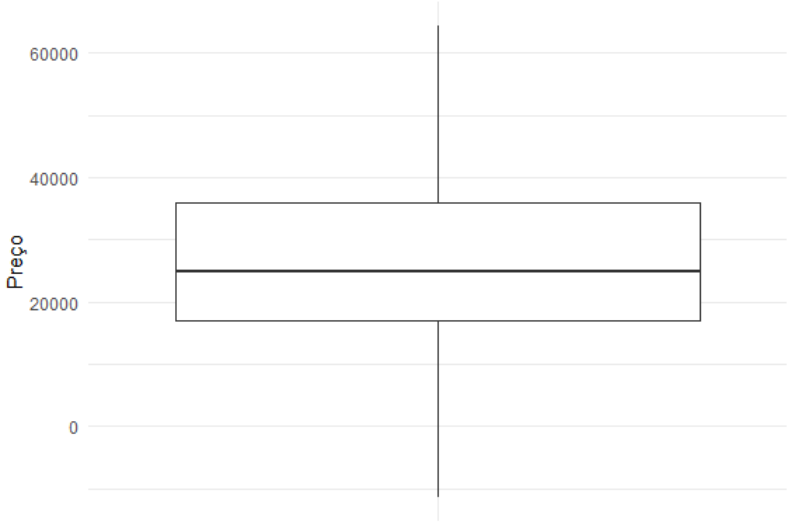

Figura 19: Boxplot de preços dos carros usados

Foram realizadas contagens para cada um das variáveis categóricas, para posteriormente os dados faltantes serem preenchidos pela categoria com maior frequência.

used_cars.R

```
178 count = lapply(categoric, \(x) {
179   arrange(
180     agg(
181       groupBy(
182         filter(data, isNotNull(data[[x]])), alias(data[[x]], x)),
              alias(count(data[[x]]), "n")
183       ), col = "n", decreasing = TRUE)
184   })
185
186 count = lapply(count, \(x) collect(x)) |>
187   `names<-`(categoric)
```

Um gráfico de pareto foi confeccionado para identificar as categorias que representavam cerca de 95% dos dados, transformando as categorias abaixo de 5% em um único grupo, denomidado "*Others*".

used_cars.R

```
189 pareto = function(dados){
190  dados$prop = dados$n/sum(dados$n)
191  dados$cumsum_ = cumsum(dados$n)
192  dados$dados = ordered(dados[,1], dados[,1])
193
194  label = sprintf("%.0f%%", 100 * dados$cumsum_ / sum(dados$n))
195
196  ggplot(dados, aes(x = dados)) +
197    geom_bar(aes(y = n), stat = "identity") +
198    geom_point(aes(y = cumsum_)) +
199    geom_path(aes(y = cumsum_, group = 1)) +
200    geom_text(aes(y = cumsum_), label = label, vjust = -1.7) +
201    scale_y_continuous("Frequência",
202                       sec.axis = sec_axis(
203                         ~ . / sum(dados$n),
204                         labels = scales::percent,
205                         name = "Acumulado"
206                       )) +
207    coord_cartesian(clip = 'off') +
208    theme_minimal() +
209    theme(plot.margin = margin(100, 10, 10, 10, "pt"),
210          axis.title.y.left = element_text("Frequência"),
211          axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1.25),
212          axis.title.x.bottom = element_blank())
213 }
214
215 pareto(head(count$body_type, 9))
```

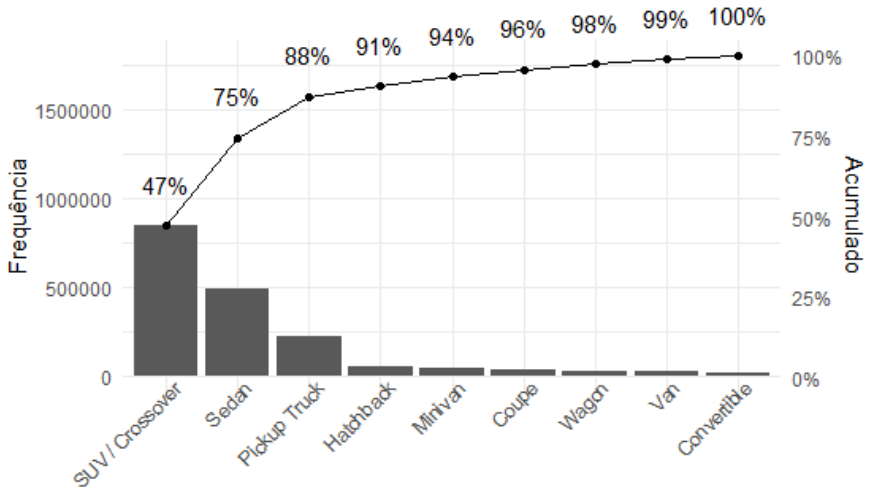

Figura 20: Frequência dos tipos de chassis do carro

Aplicando a regra para todas as colunas categóricas. Também é importante ressaltar que algumas colunas que deveriam conter apenas variáveis dicotômica, na verdade continham diversas categorias, nesses casos os valores foram substituídos por dados faltantes.

used_cars

```
216 data = transform(data, body_type = ifelse(data$body_type %in%
        {{head(count$body_type, 9)[, 1]}}, data$body_type, NA))
217
218 city_95 = count$city$city[cumsum(count$city$n)/sum(count$city$n) <= 0.95]
219 data = transform(data, city = ifelse(data$city %in% {{city_95}}, data$city,
        "Others"))
220
221 engine_type_95 =
        count$engine_type$engine_type[cumsum(count$engine_type$n)/sum(count$engine_type$n)
        <= 0.95]
222 data = transform(data, engine_type = ifelse(data$engine_type %in%
        {{engine_type_95}}, data$engine_type, "Others"))
223
224 fuel_type_95 =
        count$fuel_type$fuel_type[cumsum(count$fuel_type$n)/sum(count$fuel_type$n)
        <= 0.95]
```

```
225 data = transform(data, fuel_type = ifelse(data$fuel_type %in% {{fuel_type_95}},
        data$fuel_type, "Others"))
226
227 listing_color_95 =
        count$listing_color$listing_color[cumsum(count$listing_color$n)/sum(count$listing_color
        <= 0.95]
228 data = transform(data, listing_color = ifelse(data$listing_color == "UNKNOWN",
        NA, data$listing_color))
229 data = transform(data, listing_color = ifelse(data$listing_color %in%
        {{listing_color_95}}, data$listing_color, "Others"))
230
231 make_name_95 =
        count$make_name$make_name[cumsum(count$make_name$n)/sum(count$make_name$n)
        <= 0.95]
232 data = transform(data, make_name = ifelse(data$make_name %in% {{make_name_95}},
        data$make_name, "Others"))
233
234 data = transform(data, transmission = ifelse(data$transmission %in% c("A",
        "CVT", "M", "Dual Clutch"), data$transmission, NA))
235
236 data = transform(data, wheel_system = ifelse(data$wheel_system %in% c("FWD",
        "AWD", "4WD", "4X2"), data$wheel_system, NA))
237
238 data = transform(data, year = ifelse(data$year > 2021, NA, data$year))
239
240 data = transform(data, fleet = ifelse(data$fleet %in% c("True", "False"),
        data$fleet, NA))
241 data = transform(data, has_accidents = ifelse(data$has_accidents %in% c("True",
        "False"), data$has_accidents, NA))
242 data = transform(data, isCab = ifelse(data$isCab %in% c("True", "False"),
        data$isCab, NA))
243 data = transform(data, is_new = ifelse(data$is_new %in% c("True", "False"),
        data$is_new, NA))
```

3.4 Imputação de dados

A presença de dados faltantes em bancos de dados impossibilita o treinamento de modelos de aprendizado de máquina, por tanto utilizou-se técnicas simples de imputação para lidar com a falta de informação de alguns indivíduos. Para as variáveis categóricas, a categoria com maior frequência era aplicada aos dados faltantes, já para as variáveis numéricas utilizou-se a mediana das observações.

used_cars.R

```
246 most_frequent = lapply(count, \(x) x[1,1]) |>
247   `names<-`(categoric)
248
249 data = fillna(data, most_frequent)
250
251 medians = lapply(numeric, \(x) alias(percentile_approx(data[[x]], percentage =
        .5), x))
252 medians = collect(select(data, medians))
253
254 data = fillna(data, as.list(medians))
```

3.5 Treinamento e avaliação do modelo

Com os dados faltantes devidamente imputados, separou-se os dados em 75% para treino e 25% para teste. Os dados de treino foram aplicados a um modelo de árvore de decisão. Com o modelo treinado foi realizada a predição dos valores do teste.

used_cars.R

```
258 train_test_split = randomSplit(data, c(75, 25), 20210908)
259
260 train = train_test_split[[1]]
261 test = train_test_split[[2]]
262
263 model = spark.decisionTree(train, price ~ .)
264
265 predictions = predict(model, test)
266 head(predictions)
267
```

```
y_avg = collect(agg(predictions, avg(predictions$label)))[[1]]
df =
  agg(predictions,
      sq_res = sum((predictions$label - predictions$prediction)^2),
      sq_tot = sum((predictions$label - y_avg)^2))

SSR = collect(select(df, df$sq_res))[[1]]
SST = collect(select(df, df$sq_tot))[[1]]

Rsq = 1 - (SSR/SST)
sprintf("R: %.3f", Rsq)

write.ml(model, "./model/decisiontree_cars")
```

Foi obtido um R^2 de 0,604, outros modelos podem ser treinados utilizando diferentes técnicas ou variáveis explicativas com o objetivo de melhorar os resultados aqui obtidos. Porém, para o intuito deste trabalho, que busca apenas apresentar o SparkR e suas funções, considerou-se que o valor obtido foi satisfatório.

4 Conclusões

Durante a confecção deste trabalho o Spark demonstrou-se uma incrível ferramenta para tratar grandes volumes de dados com eficiência e simplicidade. No entanto, sua *API* para R, o SparkR, possui muitas limitações em diversas frentes como, criação de *pipelines* de aprendizado de máquina, métodos de avaliação de modelos, entre outros, estando muito atrás das *API*s em outras linguagens, como o PySpark, biblioteca da linguagem de programação Python. Além disso, a documentação oficial do SparkR por vezes não é clara e carecem exemplos, o que dificulta sua utilização.

O *SparkR* também possui inúmeros problemas no nome de suas funções e argumentos por não possuírem um padrão. Tomando as funções **isNotNull**, **as.DataFrame**, **spark.lm** e **percentile_approx**, como exemplos observa-se que em alguns casos é utilizada a notação *camel case*, em outros momentos letras minúsculas separadas por sublinhado ou até mesmo a presença de prefixos. As funções do *SparkR*, em alguns casos, também compartilham nomes com funções do R base.

O *sparklyr* é uma solução que facilita a instalação, manipulação e visualização de dados, além da aplicação de modelos com o Spark. Também permite que o *SparkDataFrame* tenha uma compatibilidade com o pacote *dplyr* entre outras ferramentas do tidyverse, tornando-se uma alternativa mais simples a usuários já familiarizados com a sintaxe do pacote *dplyr*.

Portanto, apesar da grande capacidade do Spark em lidar com dados, sua utilização através do pacote *SparkR* possui limitações em diversos aspectos.

Referências

1 Apache. *Welcome! - The Apache Software Foundation*. [S.l.]. Disponível em: <https://www.apache.org/>.

2 DEAN, Jeffrey; GHEMAWAT, Sanjay. *MapReduce: Simplified Data Processing on Large Clusters*. [S.l.]: USENIX Association OSDI '04: 6th Symposium on Operating Systems Design and Implementation.

3 R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2014. Disponível em: <http://www.R-project.org/>.

4 KOLBERG, Wagner. *Simulação e Estudo da Plataforma Hadoop MapReduce em Ambientes Heterogêneos*. [S.l.]: Universidade Federal do Rio Grande do Sul, 2010.

5 SALLOUM RUSLAN DAUTOV, Xiaojun Chen Patrick Xiaogang Peng Salman; HUANG, Joshua Zhexue. *Big data analytics on Apache Spark*. [S.l.]: Springer International Publishing Switzerland 2016, 2016.

6 LURASCHI KEVIN KUO, Edgar Ruiz Javier. *Mastering Spark with R: The Complete Guide to Large-Scale Analysis and Modeling*. [S.l.]: O'Reilly Media, 2019.

7 FOUNDATION, The Apache Software. *SparkR (R on Spark)*. Disponível em: <https://spark.apache.org/docs/latest/sparkr.html>. Acesso em: 27/07/2021.

8 RSTUDIO. *sparklyr: R interface for Apache Spark*. Disponível em: <https://spark.rstudio.com/>. Acesso em: 11/12/2020.

9 FOUNDATION, The Apache Software. *Documentation for package 'SparkR' version 3.0.1*. Disponível em: <https://spark.apache.org/docs/latest/api/R/index.html>. Acesso em: 27/07/2021.

10 FOUNDATION, The Apache Software. *MapReduce Tutorial*. Disponível em: <https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html>. Acesso em: 11/12/2020.

11 IBM. *What is MapReduce?* Disponível em: <https://www.ibm.com/analytics/hadoop/mapreduce>. Acesso em: 11/12/2020.

12 MENG, et al Xiangrui. *MLlib: Machine Learning in Apache Spark*. [S.l.], 2016.

13 GTA/UFJR. *Os 5 V's do Big Data*. Disponível em: <https://www.gta.ufrj.br/grad/15_1/bigdata/vs.html#:~:text=Os5V'sdoBigData&text=Apropostadeumasoluç~ao,aVeracidadeeoValor.> Acesso em: 27/07/2021.

14 ORACLE. *Big data definido*. Disponível em: <https://www.oracle.com/br/big-data/what-is-big-data/>. Acesso em: 27/07/2021.

15 WICKHAM, Hadley. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. Disponível em: <https://ggplot2.tidyverse.org>.

16 ZAHARIA, Matei et al. *Spark: Cluster Computing with Working Sets*. [S.l.], 2010. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.html>.

17 PATGIRI, Ripon; AHMED, Arif. Big data: The v's of the game changer paradigm. In: . [S.l.: s.n.], 2016.

18 WICKHAM, Hadley et al. *dplyr: A Grammar of Data Manipulation*. [S.l.], 2021. R package version 1.0.7. Disponível em: <https://CRAN.R-project.org/package=dplyr>.

19 SIMON, Phil. *Too Big to Ignore: The Business Case for Big Data*. [S.l.]: Wiley, 2015. ISBN 978-1-119-21784-8.

APÊNDICE 1 – Descrição das colunas do banco de dados

Nome	Tipo	Descrição
vin	String	Código único de identificação do veículo
back_legroom	String	Espaço para as pernas no banco traseiro
bed	String	Categoria da pick, usualmente dados vazios indicam que o veículo não é uma pickup
bed_height	String	Altura da parte aberda de pickups em polegadas
bed_length	String	Largura da parte aberta de pickups em polegadas
body_type	String	Tipo de corpo do carro, sedan, conversível, hatch, etc
cabin	String	Categoria do espaço da cabine em pickups
city	String	Cidade do anúncio
city_fueleconomy	Float	Economia de combustível no tráfego da cidade
combine_fueleconomy	Float	Média ponderada entre cityfueleconomy e highway_fueleconomy
daysonmarket	Integer	Quantidade de dias que o anúncio estava no site
dealer_zip	Integer	Código postal do vendedor
description	String	Descrição do anúncio do veículo
engine_cylinders	String	Configuração do motor
engine_displacement	Float	Mensura o volume varrido pelo cilindro por todos os pistões, incluindo as câmaras de combustão

Nome	Tipo	Descrição
engine_type	String	Tipo de motor
exterior_color	String	Cor do exterior do carro
fleet	Boolean	Indica se o carro foi parte de uma frota
frame_damaged	Boolean	Indica se o carro tem danos
franchise_dealer	Boolean	Indica se o vendedor é de uma franquia
franchise_make	String	Companhia dona da franquia
front_legroom	String	Espaço para pernas do banco da frente em polegadas
fueltankvolume	String	Capacidade do tanque de combustível do carro em galões
fuel_type	String	Tipo de combustível no qual o carro foi predominantemente abastecido
has_accidents	Boolean	Indica se o carro já se envolveu em algum acidente
height	String	Altura do carro em polegadas
highwayfueleconomy	Float	Economia de combustível em estrada em km/litro
horsepower	Float	Cavalos de potência fornecidos pelo motor
interior_color	String	Cor do interior do veículo
isCab	Boolean	Indica se o carro já foi um táxi.
is_certified	Boolean	Indica certificação do carro
is_cpo	Boolean	Carros usados certificados pelo negociador
is_new	Boolean	Indica se o carro foi lançado a menos de 2 anos da data da coleta dos dados
is_oemcpo	Boolean	Identifica se o carro usado é certificado pela fabricante
latitude	Float	Latitude da geolocalização do vendedor
length	String	Largura do carro em polegadas
listeddate	String	Dia no qual o carro foi anunciado no site
listing_color	String	Cor predominante do exterior do carro
listing_id	Integer	Código de identificação único do anúncio do site
longitude	Float	Longitude da geolocalização do vendedor
mainpictureurl	String	Url que contém a imagem do carro