Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

C. Lente

Estruturas de Dados Distribuídas para a Linguagem R

São Paulo
Dezembro de 2018

Estruturas de Dados Distribuídas para a Linguagem R

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman

São Paulo
Dezembro de 2018

Resumo

Esta monografia pretende descrever o pacote `ddR`, uma programoteca que implementa estruturas de dados distribuídas (DDSs) para a linguagem `R` de programação, e propor duas extensões para a mesma. Utilizando primitivas de paralelismo disponibilizadas pelo `ddR`, os autores implementam as duas extensões discutidas: versões concorrentes e distribuídas dos algoritmos de análise de componentes principais e *bootstrap*. Estes algoritmos também têm seu desempenho avaliado e comparado com os seus equivalentes sequenciais.

Keywords: distributed-data-structures, high-performance-computing, r-language.

Abstract

This final paper seeks to describe the `ddR` package, a programming library that implements distributed data structures (DDSs) for the `R` language, and propose two extensions to it. Using parallel primitives made available by `ddR`, the author implement the two discussed extensions: concurrent and distributed versions of the principal component analysis and bootstrap algorithms. These functions have their performance evaluated and compared to their sequential equivalents.

Keywords: distributed-data-structures, high-performance-computing, r-language.

Sumário

Capítulo 1

Introdução

Neste capítulo é realizada uma breve introdução de forma a contextualizar o resto deste trabalho. Primeiramente é apresentado o básico da linguagem `R` de programação, que pode não ser tão conhecida pela audiência desse texto. Logo em seguida há uma recapitulação da história do *hardware* da computação, focando nos avanços realizados na área do paralelismo.

1.1 A Linguagem R

A linguagem `R` de programação foi publicada pela primeira vez no ano de 1993 tendo sido desenvolvida por dois pesquisadores da Universidade de Auckland na Nova Zelândia, Ross Ihaka e Robert Gentleman[1]. O objetivo de Ihaka e Gentleman (1996) era simular a sintaxe da linguagem `S` (muito comum na época entre estatísticos) e a semântica da linguagem `Scheme` em um único pacote GNU que seria completamente *open source*.

Com o passar dos anos, o `R` foi crescendo em popularidade na comunidade estatística e de ciência de dados no geral até chegar mais recentemente ao décimo oitavo lugar do índice TIOBE (TIOBE software BV, 2018) que classifica as linguagens de programação mais utilizadas do mundo.

A semântica do `R` é bastante heterodoxa quando comparada com linguagens de programação mais conhecidas pelos programadores no geral, o que acaba sendo tanto uma vantagem quanto uma desvantagem no tocante ao crescimento da popularidade da linguagem. Algumas dessas funcionalidades serão abordadas em mais detalhes no capítulo 2, mas é suficiente dizer que é muito fácil criar programotecas para o `R` que alteram completamente o funcionamento da linguagem e até mesmo a forma como se programa nela.

```r
summary(mtcars$mpg)
#  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
# 10.40   15.43   19.20   20.09   22.80   33.90
f <- function(x, ...) { x + mean(...) }
f(1:3, 4:5)
# [1] 5.5 6.5 7.5
```

Código 1.1: *Exemplo básico de código R que destaca sua sintaxe idiossincrática. Note que comentários são demarcadas por uma cerquilha.*

[1] De acordo com próprios doutores, apesar de toda a energia gasta ao longo do projeto, eles não perderam a amizade e nem o interesse por computação.

Pacotes cujo objetivo é mudar a forma com que se programa `R` permitiram que a linguagem crescesse ao mesmo tempo em que a maioria dos códigos escritos com ela permanecessem funcionais. Outras linguagens criadas da mesma época como o `Python` passaram por processos muito mais traumáticos de evolução, com quebras de *API* e cismas na comunidade.

Entretanto essa solidez da linguagem acaba sendo uma faca de dois gumes: por um lado ela permite que o mesmo código seja mantido por muito tempo (R Core Team, 2018b) e que os programadores não precisem se renovar a cada tantos anos, mas ela também engessa o código fonte da linguagem, impedindo que ela evolua de forma dinâmica.

Na prática, o que se percebe hoje em dia é que o `R`[2] está mostrando a sua idade. Na opinião dos autores, a faceta pela qual isso fica mais claro é na programação paralela e concorrente: o `R`, diferentemente de diversas linguagens modernas como `Go` e `Rust`, não possui suporte nativo para *multithreading*. Isso quer dizer que fazer um código sequencial mais complexo ser processado em paralelo pode requerer extenso conhecimento sobre computação de alto desempenho por parte do programador.

1.1.1 Programação Paralela em R

O `R` é por design uma linguagem *singlethreaded*, ou seja, cuja execução ocorre em um único encadeamento sequencial. Somente na versão 2.14.0, com a adição do módulo `parallel`, o `R` passou a ser capaz de rodar código em paralelo.

A solução do R Core Team (2018a) para a *singlethreadness* da linguagem foi utilizar *forks*, fazendo com que a sessão principal criasse sessões filhas onde as partições da tarefa seriam executadas. Isso resolveu o problema do paralelismo em si, mas nunca foi adicionado nenhum outro módulo que permitisse controle fino de *threads* ou do escalonamento.

Para suprir essa defasagem da linguagem, a comunidade foi aos poucos criando pacotes que fornecessem as utilidades que os usuários desejavam. Partindo do `parallel` e mais alguns outros poucos pacotes, foram sendo criados mais e mais programotecas que eram capazes de realizar suas computações em paralelo.

O problema é que o usuário padrão do `R` (comumente de formação em Estatística) acaba ficando à mercê desses pacotes secundários, sendo obrigado a esperar que outros implementem ferramentas paralelizáveis para resolver os seus problemas. Eddelbuettel (2018) compilou uma lista de todos os pacotes publicados [3] do `R` que diziam respeito pelo menos tangencialmente a computação de alto desempenho e concluiu que (em meados de 2018) eles já eram quase 100.

Esses pacotes (o termo utilizado no `R` para "biblioteca"ou "programoteca") se enquadram em diversas categorias: paralelismo explícito, paralelismo implícito, aplicações paralelizadas, dentre muitas outras. Muitos deles são bastante minimalistas e oferecem versões paralelizadas de apenas um ou dois algoritmos, mas há também outros que pretendem fornecer *frameworks* completos de paralelismo para a linguagem. O ponto chave comum à grande maioria desses pacotes é que eles não têm a pretensão de interagirem uns com os outros, fazendo com que cada um acabe se tornando um ecossistema fechado cujo os usuários precisam aprender do zero a utilizar.

Nessa paisagem pulverizada e muitas vezes hostil para o usuário médio nota-se a necessidade da criação de arcabouços ferramentais que favoreçam a usabilidade, a interoperabilidade e, acima de tudo, a extensibilidade. Foi com isso em mente que Ma *et al.* (2016b) criaram o

[2] Na data de escrita, o R está na sua versão 3.4.4 – "Someone to Lean On"

[3] Diferentemente de outras linguagens, o R possui um órgão central que revisa e publica pacotes. As programotecas sancionadas pelo CRAN são periodicamente testadas e também são mais fáceis de instalar do que aquelas disponibilizadas somente via Git.

pacote `ddR`.

O funcionamento e a inovação tecnológica por trás do `ddR` serão abordados no capítulo 2 deste texto. Antes se faz necessário contextualizar a computação de alto desempenho como um todo e o seu atual maior motivador, o *machine learning*.

1.2 História da Computação Paralela

A história da computação moderna começa no final da década de 1950 com o advento dos circuitos integrados (também conhecidos como *microchips*). Esses circuitos foram utilizados primeiramente em sistemas embarcados por gigantes da tecnologia como a NASA, mas poucos anos depois nos meados da década de 1960 essa tecnologia já começava a aparecer em computadores.

A primeira geração de computadores utilizava tubos de vácuo como centro de processamento, enquanto a segunda utilizava transistores. Com a adoção dos circuitos integrados como núcleo do computador é que começa o que hoje em dia é chamada de a terceira geração de computadores.

Foi nessa época que o cofundador da Fairchild Semiconductor e da Intel, Gordon Moore, publicou um artigo em que ele levantou a hipótese de que a cada dezoito meses dobraria o número de componentes por circuito integrado. Essa máxima, hoje em dia chamada de Lei de Moore, acabou tornando-se uma profecia auto-cumprida: as empresas de silício acabaram a utilizando como meta anual, mantendo-a válida durante as décadas seguintes.

Conforme foi aumentando a capacidade de processamento dos circuitos integrados e foi crescendo a necessidade de miniaturização mais efetiva, o próximo passo na escala evolutiva se tornava cada vez mais viável: integração em larga escala (ou LSI, do inglês *large-scale integration*).

Esses avanços culminaram no advento em 1971 do primeiro microprocessador comercial: o Intel 4004. Diferentemente dos seus predecessores, os microprocessadores eram compostos por uma unidade de processamento central (CPU) completamente restrita a uma única placa de circuito.

Com esse novo paradigma de *hardware* começou a verdadeira revolução dos computadores: miniaturização, barateamento e produção em larga escala. A quarta geração de computadores foi a com o maior impacto até hoje, em grande parte graças ao advento do microprocessador.

Durante os próximos quase 20 anos, os fabricantes de *chips* focaram praticamente todos os seus esforços em aumentar o número de transistores nos circuitos que compunham as CPUs e em melhorar a frequência dos ciclos de processamento (geralmente chamado de *clock*). Essas eram as formas mais fáceis de aumentar a potência de um computador, pois elas permitiam que a CPU processasse mais instruções mais rapidamente.

Mas esse tipo de estratégia raramente pode ser utilizada por muito tempo. Logo percebeu-se que os limites físicos dos componentes não seriam superados tão facilmente pelos avanços tecnológicos como se havia pensado anteriormente. De acordo com Herlihy e Shavit (2011), foi por isso que na década de 1980 os fabricantes de computadores começaram a fazer experimentos com paralelismo de instruções.

Nos meados da década de 1980 as empresas de hardware já estavam comercializando processadores que implementavam essa nova técnica de paralelização. Sem paralelismo de instruções, os computadores estavam limitados a no máximo uma instrução de código por ciclo do *clock*, mas agora eles eram capazes de processar mais de uma instrução ao mesmo tempo.

Os processadores capazes de rearranjar e reagrupar instruções de forma que o seu processamento concomitante não atrapalhasse o resultado do programa eram chamados de superescalares (em contraponto com os computadores escalares que só conseguiam escalonar no máximo uma instrução por ciclo). Os processadores superescalares foram a arquitetura dominante no mercado até meados da década de 1990, quando um novo avanço tecnológico deixaria essa abordagem obsoleta: paralelismo a nível de tarefas.

Apesar de ser o próximo passo lógico na evolução dos sistemas paralelos, a implementação de paralelismo a nível de tarefas depende de um comportamento muito mais proativo do processador. A partir do momento em que os detalhes de implementação haviam sido resolvidos, os processadores passaram a ser capazes de distribuir múltiplas instruções provenientes de múltiplas fontes a cada ciclo, enquanto os processadores superescalares eram capazes de distribuir instruções provenientes de apenas uma só fonte.

Com o paralelismo a nível de tarefa passou a ser possível o que hoje chamamos de concorrência via *multithreading*, ou seja, a emulação de múltiplos núcleos de processamento com uma só CPU. Em processadores com *multithreading* há múltiplos fluxos de instruções sendo processados concomitantemente pela CPU.

Para que houvesse um ganho ainda maior de desempenho os múltiplos núcleos em um processador precisavam deixar de ser emulados e passar a ser uma realidade, o que chamamos de arquitetura *multicore*. Essa é a infraestrutura mais comum atualmente, de forma que já é possível comprar celulares de ponta que possuem processadores *quad-core*, a saber, processadores com quatro núcleos.

1.3 Aprendizado de Máquina

Nos dias de hoje, aumentar o desempenho dos computadores continua sendo uma área ativa de pesquisa. Com os ganhos alcançados ao longo das últimas décadas, técnicas de análise de dados e modelagem estatística que dependem de *big data* ou aprendizado de máquina passaram a estar ao alcance do usuário comum.

Apesar de o campo estar em desenvolvimento desde os anos 1950, a grande disseminação do *machine learning* começou depois dos anos 2000. Por causa do barateamento de *hardware* poderoso, da disponibilidade de amplo espaço para armazenamento de dados e da criação de múltiplos *frameworks open source* para facilitar o uso dos algoritmos de aprendizado, hoje em dia qualquer empresa que esteja minimamente envolvida com dados usa essas técnicas no seu dia a dia.

A grande questão é que ainda existe um grau elevado de dispersão no ecossistema de aprendizado de máquina. Uma rápida pesquisa na internet em busca de ferramentas para *machine learning* já devolve dezenas de resultados: `CNTK`, `H2O`, `scikit-learn`, `Spark MLlib`, `TensorFLow`, `Torch`, dentre outros.

Entrar em detalhes sobre as funcionalidades, vantagens e desvantagens de estes *frameworks* está além do escopo deste trabalho, mas é importante ressaltar a utilidade e o impacto que as técnicas de aprendizado de máquina vêm tendo atualmente.

Apesar de englobar uma variedade enorme de algoritmos e teorias, aprendizado de máquina pode ser descrito como o uso de métodos estatísticos com o objetivo de fazer com que sistemas computacionais "aprendam", ou seja, melhorem progressivamente na execução de uma tarefa através de dados (sem ser explicitamente programados).

Por ser pouco específica, essa definição acaba sendo aplicada desde a algoritmos de regressão linear até a assistentes de inteligência artificial, mas isso não desqualifica os impressionantes resultados que puderam ser alcançados por esse campo de pesquisa. Atualmente algoritmos de *machine learning* são utilizados para predizer quais filmes serão preferidos por

um usuário, modelar mercados financeiros, identificar objetos em imagens em microssegundos, dentre muitas outras aplicações.

Mas, entre possuir o arcabouço teórico das técnicas de aprendizado de máquina e de fato colocar seus modelos em produção, existe um grande abismo. Muitos desenvolvedores acabam perdendo-se nesse ecossistema anárquico e pulverizado de bibliotecas, dificultando a usabilidade e a propagação do conhecimento.

Usuários não acostumados com muitas linguagens de programação acabam ficando limitados a um punhado de *frameworks* pouco otimizados para o *hardware* altamente paralelizável que está ao alcance de qualquer um hoje em dia.

1.4 O Projeto

Depois da contextualização apresentada nas seções anteriores, é possível notar um tema comum entre o ecossistema do `R` de programação paralela e o ecossistema mundial de *frameworks* de aprendizado de máquina: falta de coesão. Como será discutido mais a frente, a programoteca `ddR` pretende ajudar com ambos problemas através de uma solução simples mas engenhosa.

O objetivo deste trabalho é, portanto, auxiliar o projeto `ddR` através de extensões ao seu repertório de algoritmos. A primeira extensão é uma função de análise de componentes principais e, a segunda, uma função de *bootstrap*.

O capítulo 2 fala sobre os conceitos fundamentais necessários para compreender as extensões realizadas: estruturas de dados distribuídas e o pacote `ddR` em si. Logo a seguir, no capítulo 3, são apresentadas as extensões realizadas pelos autores e, no capítulo 4, as mesmas têm seus desempenhos avaliados.

Por fim, o trabalho fecha com uma recapitulação no capítulo 5. Há também dois apêndices (A e B) para estender alguns pontos auxiliares ao texto.

Capítulo 2

Fundamentos

Neste capítulo são abordados os tópicos fundamentais para compreender os trabalhos desenvolvidos. A primeira seção aborda estruturas de dados concorrentes e distribuídas, enquanto a segunda aborda o pacote `ddR` mais diretamente. O capítulo como um todo serve como motivados para os desenvolvimentos apresentados no capítulo 3.

2.1 Estruturas de Dados Concorrentes

Uma estrutura de dados concorrente é aquela que pode ser acessada e manipulada concomitantemente por múltiplos *threads*. Como apontam Moir e Shavit (2001), a proliferação de máquinas multiprocessador trouxe consigo a necessidade de estruturas de dados concorrentes bem desenhadas dado que estes sistemas geralmente se comunicam através de estruturas de dados em memória compartilhada.

A maior dificuldade associada a estruturas de dados concorrentes (doravante abreviadas como "EDCs") está relacionada à própria concorrência: como os *threads* executam simultaneamente em diferentes núcleos ou processadores, eles estão sujeitos ao escalonamento do sistema operacional de forma que é necessário considerar todas as computações como completamente assíncronas, ou seja, com as ações dos diferentes *threads* intercaladas arbitrariamente.

Outro ponto importantíssimo do desenho de EDCs é o equilíbrio entre corretude e desempenho. Modificar um algoritmo com o objetivo de aumentar a sua eficiência temporal pode dificultar muito a sua implementação e, consequentemente, tornar sua corretude quase impossível de garantir.

O ponto mais importante da corretude de uma EDC (ou qualquer outro algoritmo paralelo) é a garantia da exclusão mútua; se um estrutura de dados ou algoritmo concorrente não consegue garantir que *threads* concorrentes não entrarão em suas seções críticas ao mesmo tempo, então sua corretude não pode ser garantida. É por isso que uma das decisões mais relevantes no desenho de uma EDC é o nível de granularidade de exclusão mútua.

Ellen e Brown (2016) apontam que existem dois polos opostos no tocante a *locks* (garantidores de exclusão mútua): *locks* globais e *locks* granulares. Com a primeira, apenas o processo segurando a chave para a EDC pode acessá-la; essa técnica garante a exclusão mútua com tranquilidade, mas é muito restritiva e portanto acaba tendo um impacto negativo no desempenho. Já a segunda envolve múltiplas *locks* protegendo diferentes partes da estrutura de dados, maximizando a concorrência; neste caso o desempenho não é necessariamente afetado, mas garantir a *liveness* e a *safety* se tornam tarefas complexas.

2.1.1 Estruturas de Dados Distribuídas

Similarmente às estruturas de dados concorrentes, as estruturas de dados distribuídas (doravante "EDDs") são objetos que podem ser acessados simultaneamente por múltiplos fluxos de execução. Neste caso entretanto, a estrutura estará sendo dividida ao longo de múltiplas máquinas distribuídas em um *cluster* ou uma *cloud*.

Esse tipo de estrutura de dados é muito menos comum do que os seus pares concorrentes, mas, com o avanço das tecnologias baseadas em sistemas distribuídos, ele tem ganhado cada vez mais espaço. Gribble *et al.* (2000) foram uns dos primeiros a utilizar esse modelo computacional quando criaram uma forma de simplificar a construção de serviços de Internet baseados em *clusters*.

É importante destacar que não existe um consenso a respeito das nomenclaturas "concorrente"e "distribuída"quando aplicadas a estruturas de dados; muitos autores usam o termo EDD para se referir ao que aqui são chamadas de EDCs e vice-versa. Por questões de implementação, uma EDD é necessariamente uma EDC e, portanto, é comum chamar ambas as variantes de estruturas de dados concorrentes. Neste trabalho também será usado outro termo não padronizado para se referir ao conjunto formado pelos dois tipos de estruturas: "estruturas de dados paralelas".

2.1.2 Implementações Mais Comuns

Provavelmente uma das implementações de estruturas de dados concorrentes mais conhecidas atualmente vem do `Spark`, um *framework open-source* para computação distribuída multipropósito. Recentemente o `Spark` tem crescido muito em popularidade por múltiplos motivos: velocidade, integração com o `Hadoop`, latência mínima, variedade de linguagens conectadas e dedicação extensa a aprendizado de máquina[1].

Para executar computações em *cluster* o `Spark` faz uso de uma abstração principal chamada RDD (de *resilient distributed dataset*, traduzido livremente como "conjunto de dados distribuído"). De modo geral, toda aplicação em `Spark` consiste de um programa denominado *driver* que roda a função principal do usuário e executa diversas operações paralelas em um *cluster*; nesse contexto o RDD é a coleção de elementos particionada ao longo dos múltiplos nós do *cluster* na qual pode-se operar em paralelo. RDDs são criados a partir de um arquivo no sistema de arquivos do `Hadoop` (ou em qualquer outro sistema de arquivos com suporte para `Hadoop`) ou a partir de uma coleção do `Scala` já existente no programa *driver*. Um usuário também pode fazer com que um RDD seja persistido em memória, permitindo que ele seja reutilizado com eficiência em diversas operações paralelas.

O modelo para o RDD foi criado por Zaharia *et al.* (2010) como uma alternativa a modelos de computação distribuída baseados em fluxos de dados acíclicos, inadequados para aplicações que reutilizassem um único conjunto de dados em múltiplas operações paralelas (aplicações estas que já eram muito comuns por causa da ascensão de algoritmos de *machine learning* e ferramentas de análise iterativos).

Arquiteturalmente, um RDD não passa de uma coleção de objetos *read-only* particionada ao longo de um conjunto de máquinas que pode ser reconstruída caso uma destas partições seja perdida. Assim, um RDD tem as seguintes propriedades:

- Resiliência: um RDD pode ser recriado em caso de falha;
- Distribuição: um RDD pode ser particionado e distribuído ao longo de múltiplos nós;

[1] Destacam-se a MLlib do próprio Spark e a biblioteca externa de conexão com o H2O apropriadamente chamada de Sparkling Water.

- Imutabilidade: um RDD não pode ser modificado, somente transformado para gerar outro RDD;
- Preguiça: um RDD representa o resultado de uma computação, mas não necessariamente desencadeia nenhuma computação (assim é possível descrever uma cadeia completa de operações sem executar nada) e
- Estaticidade: um RDD é estaticamente tipado, então há aumento de assertividade.

2.2 O Pacote ddR

Por ser uma ferramenta popular de análise de dados, diversos sistemas distribuídos possuem uma *API* voltada para o `R`. O resultado disso é que cientistas de dados precisam aprender a usar múltiplas interfaces como `RHadoop`, `SparkR`, `ScaleR` e `Distributed R` de modo a ter acesso a aplicações distribuídas. Infelizmente estas interfaces não são padronizadas e muitas vezes são difíceis de aprender, fazendo com que programas em R escritos em um *framework* não funcionem em outro mesmo que tenha havido um esforço por parte das infraestruturas de implementar os mesmos algoritmos distribuídos de aprendizado de máquina.

A utilidade destas diferentes interfaces é indiscutível, pois elas permitem que cientistas de dados continuem programando no familiar `R` ao mesmo tempo em que trabalham com bases de dados muito grandes. O problema é que cada um destes *backends* expõe uma *API* diferente para o `R`, impedindo que as aplicações programadas em um *backend* possam utilizar algum outro. Muitas interfaces inclusive expõem detalhes de baixo nível dos *backends*, fazendo com que sejam difíceis de aprender.

O principal motivo para esse cenário fragmentado é a ausência de qualquer forma padronizada de se fazer programação paralela e distribuída no `R`. No tocante a bases relacionais, por exemplo, a semântica `SQL` unificou a forma como se manipula dados, mas não houve muito esforço por parte das comunidades de ciência de dados e aprendizado de máquina de criar uma padronização similar para análise de dados avançada. Apesar de a comunidade `R` ter contribuído em milhares de bibliotecas, há poucas versões paralelas ou distribuídas destes algoritmos.

Com o objetivo de criar uma interface simples e padronizada para computação distribuída em `R`, Ma *et al.* (2016b) desenvolveram a programoteca `ddR`. Os autores deste *framework* esperavam que esta nova tecnologia fomentasse contribuições em aplicações paralelas de aprendizagem de máquina do `R` sem que os usuários tenham medo de ficarem presos a uma plataforma específica.

Havia dois principais desafios ao elaborar este sistema padronizado de análise avançada de dados. Primeiramente o *framework* precisaria implementar uma interface simples o suficiente para ser utilizada com facilidade por cientistas de dados sem treinamento formal em computação paralela, além de ser minimamente flexível para expressar diversas tarefas analíticas. Em segundo lugar, as aplicações expressas neste sistema precisariam ter bom desempenho já que um problema comum em interfaces genéricas é o alto *overhead*.

Para solucionar estes problemas, o `ddR` define três estruturas de dados distribuídas que espelham estruturas de dados básicas do `R`: `dlist`, `darray` e `dframe`, equivalentes a `list`, `matrix` e `data frame` respectivamente. Com estes três contêineres, o cientista de dados tem uma forma intuitiva e familiar de particionar conjuntos de dados massivos para que possam ser utilizados em aplicações paralelas e distribuídas.

2.2.1 Dmapply

Para manipular as estruturas de dados distribuídas, os autores do `ddR` (Ma *et al.*, 2016a), criaram a `dmapply()`. Esta é uma primitiva de programação com semântica funcional para que haja garantia de ausência de efeitos colaterais, facilitando a paralelização da aplicação. Além disso, o operador também é multivariado, permitindo que o programador o utilize para trabalhar com múltiplos conjuntos de dados ao mesmo tempo.

A utilidade do `dmapply()` vem do fato de que ele pode expressar diferentes padrões de computação e comunicação que são comuns em tarefas de análise de dados. Por exemplo, programadores podem facilmente expressar computações comumente denominadas "vergonhosamente paralelas"(Herlihy e Shavit, 2011), onde uma função opera em cada partição dos dados sem que haja necessidade de mover nenhum dado. Também é possível expressar situações onde certas estruturas de dados (ou partes delas) são transmitidas para todas as computações. Um usuário pode até expressar padrões em que as computações em um nó servo recebem dados de qualquer outro subconjunto de servos, assim implementando diferentes tipo de agregação.

Na publicação original, os autores demonstram que esta primitiva consegue inclusive implementar algoritmos que ajustam um modelo de *query* estatístico e o popular paradigma `MapReduce`.

O pacote `ddR` até reimplementa algumas funções padrões do `R` com a primitiva `dmapply()`, permitindo que os programadores não precisem se preocupar em aprender novas ferramentas para tirar proveito das vantagens do `ddR`. Por exemplo, programadores que usam o `SparkR` ou o `HPE Distributed R` podem utilizar a função distribuída `colMeans()` mesmo que o respectivo *backend* não tenha uma implementação nativa daquela função. Esses operadores também servem de incentivo para que os desenvolvedores de *backends* passem a integrar o `ddR` e ajudem com os esforços de padronização.

Dentre estas funções reimplementadas estão alguns do principais algoritmos de *machine learning* que realizam operações de agrupamento de dados, classificação e regressão. De acordo com as avaliações de Ma *et al.* (2016b), os algoritmos possuem desempenho muito bom; em um único servidor, um regressão com o *backend* `parallel` em 12 milhões de linhas e 50 colunas converge em 20 segundos, enquanto a análise de agrupamento de 1,2 milhões de pontos de 500 dimensões demora apensa 7 segundos por iteração. Os algoritmos do `ddR` têm desempenho parecido com, e às vezes melhor que, os seus equivalentes de produtos como `H2O` e `Spark MLlib` (cujos algoritmos não têm portabilidade). Em ambientes distribuídos usando o `HPE Distributed R` como *backend*, a regressão por k-médias do `ddR` apresenta escalabilidade quase linear e pode processar facilmente gigabytes de dados. Estes resultados mostram que usuários podem extrair grandes benefícios do uso de uma interface padronizada para expressar aplicações e ainda obter desempenho similar às implementações não-portáteis.

No código 2.1 é possível observar um exemplo de como é simples utilizar a interface do `ddR`. Na linha 11 há, para propósitos demonstrativos, a aplicação de um modelo *random forest* padrão (Liaw e Wiener, 2002) no conjunto de dados `Boston` (Venables e Ripley, 2002). As aplicações das versões paralela e distribuída do `ddR` do *random forest* são idênticas à padrão, necessitando apenas das fases de registro dos *backends*. A partir de uma avaliação de desempenho feita com a biblioteca `microbenchmark` (Mersmann, 2018), a aplicação paralela do *random forest* foi 55% mais veloz do que a aplicação padrão mesmo de não tendo necessitado nenhuma modificação.

```r
# Bibliotecas utilizadas
library(ddR)
library(randomForest.ddR)
library(distributedR.ddR)
library(randomForest)

# Dados utilizados
Boston <- MASS::Boston

# Aplicação de random forest sequencial
randomForest(medv ~ ., Boston)

# Registrar backend parallel
useBackend("parallel", nExecutor = 4)

# Aplicação paralela de random forest
drandomForest(medv ~ ., Boston)

# Registrar backend distributedR
useBackend("distributedR")

# Aplicação distribuída de random forest
drandomForest(medv ~ ., Boston)
```

Código 2.1: *Exemplo de código padrão R e código escrito com o ddR. A versão paralela é 55% mais rápida que a sequencial em um dataset de 100 linhas.*

Capítulo 3

Desenvolvimentos

Neste capítulo são apresentadas as extensões realizadas no pacote `ddR`. Primeiro é feita uma contextualização geral sobre o desenvolvimento e depois são abordados os novos algoritmos adicionados ao repertório do `ddR`: `dprcomp()` e `dboot()`.

3.1 Extensão do ddR

Em sua versão mais recente, o `ddR` conta com quatro implementações prontas de algoritmos de aprendizado de máquina: modelos lineares generalizados (GLMs), k-médias, florestas aleatórias e *pagerank*. Apesar de estes serem algoritmos bastante comuns nas aplicações de ciência de dados, ainda há uma larga seleção de algoritmos extremamente populares que um usuário precisaria implementar por conta própria para tirar vantagem da infraestrutura do `ddR`. Como discutido anteriormente, é vantajoso ter funções pré-prontas à disposição do usuário porque muitas vezes, no tocante à linguagem `R`, ele ou ela provavelmente virá de uma área diferente da computação e pode ter dificuldades de adaptar algum algoritmo complexo de aprendizado de máquina para o contexto paralelo.

O objetivo principal deste trabalho foi portanto estender o `ddR` de forma que ele disponibilizasse mais algumas ferramentas para os usuários. Diversos algoritmos foram considerados para serem implementados: modelos aditivos generalizados (GAMs), máquinas de vetores de suporte (SVMs), dentre outros. A métrica para escolher os algoritmos foi uma combinação da utilidade do mesmo no mundo real e o quão difícil seria desenvolver seu código; desta forma, foram implementadas versões concorrentes e distribuídas de análises de componentes principais (PCA) e *bootstrap*.

A implementação destas funções não poderia, entretanto, ser feita de qualquer forma. Para maximizar a compatibilidade das funções do `ddR` com código já existente e facilitar o aprendizado dos usuários, é essencial que elas tivessem os mesmos argumentos e saídas que as suas versões sequenciais. Essa foi a parte consideravelmente mais difícil do projeto porque às vezes era praticamente impossível compatibilizar a saída dos algoritmos paralelos sem perder desempenho.

Outro detalhe de implementação importante é que cada função deveria ser desenvolvida em uma programoteca separada. Uma das decisões mais relevantes dos autores do `ddR` foi de manter no pacote principal apenas as funções essenciais do *framework* (estruturas de dados e primitivas de paralelização); desta forma, o usuário deve instalar programotecas separadas para ter acesso aos algoritmos de aprendizado de máquina. Essa exigência adiciona um pouco de complexidade ao processo de desenvolvimento, mas é extremamente vantajosa porque permite que qualquer um crie novas funções para o `ddR` sem que ela necessariamente seja integrada ao repositório oficial dos autores.

Depois de avaliados estes requisitos, os autores também decidiram que seria vantajoso desenvolver testes unitários para cada um dos pacotes de modo a garantir a assertividade das funções desenvolvidas diretamente no fluxo de integração contínua. Para desenvolver os testes foi utilizado o motor desenvolvido por Wickham (2011) denominado `testthat`, o mais utilizado pela comunidade `R`.

3.1.1 Dprcomp

O algoritmo de análise de componentes principais, desenvolvido por Pearson (1901), é um procedimento estatístico que usa transformações ortogonais para converter um conjunto de pontos de variáveis possivelmente correlacionadas em um conjunto de valores linearmente independentes denominados componentes principais. O uso mais comum deste método é reduzir a dimensionalidade de dados que possuam muitas variáveis para aceleração da de tarefas de modelagem em troca da interpretabilidade dos modelos.

Mais formalmente, se há n observações em p variáveis, então o número de componentes principais distintos é $\min(n-1, p)$. Essa transformação é definida de modo que o primeiro componente principal tem a maior variância possível (capturando o máximo da variância dos dados) e cada componente sucessivo tem a maior variância possível com a restrição de que ele precisa ser ortogonal aos componentes anteriores. Os vetores resultantes (cada um sendo uma combinação linear das variáveis e contendo n observações) são uma base linearmente independente.

Seja $\mathbf{X}$ uma matriz $n \times p$ $(n > p)$ e $\mathbf{S}$ a matriz de covariância de $\mathbf{X}$, obtida pela decomposição espectral

$$n\mathbf{S} = \mathbf{V}\mathbf{\Lambda}^2\mathbf{V}^T$$

Onde $\mathbf{V}$ é a matriz dos valores singulares à direita de $\mathbf{X}$ e $\mathbf{\Lambda}^2 = \text{diag}(\lambda_1^2, \lambda_2^2, \ldots, \lambda_p^2)$ é a matriz diagonal do quadrado dos autovalores ordenados de maior para menor. Estas matrizes podem, por sua vez, ser obtidas a partir da decomposição em valores singulares

$$(\mathbf{I}_n - n^{-1}\mathbf{1}\mathbf{1}^T)\mathbf{X} = \mathbf{U}\mathbf{\Lambda}\mathbf{V}^T$$

Onde $(\mathbf{I}_n - n^{-1}\mathbf{1}\mathbf{1}^T)$ é o operador de centralização de colunas e $\mathbf{U}$ é a matriz dos valores singulares à esquerda de $\mathbf{X}$.

Como pode ser interessante extrair apenas os k componentes principais de maior variância, define-se $\widetilde{\mathbf{V}}$ sendo as primeiras k colunas de $\mathbf{V}$. Tem-se assim

$$\mathbf{Z} = (\mathbf{I}_n - n^{-1}\mathbf{1}\mathbf{1}^T)\mathbf{X}\widetilde{\mathbf{V}}^T$$

Sendo $\mathbf{Z}$ os k primeiros componentes principais de $\mathbf{X}$ centralizada em suas colunas.

A chave para a paralelização deste algoritmo é o fato de que pode-se calcular a matriz de covariância global a partir de covariâncias locais. Seja $\mathbf{X}$ particionada ao longo de suas linhas em s partições

$$n\mathbf{S} = \sum_{i=1}^{s} \mathbf{V}_i\mathbf{\Lambda}_i^2\mathbf{V}_i^T + \sum_{i=1}^{s} n_i(\bar{\mathbf{x}}_i - \bar{\mathbf{x}})(\bar{\mathbf{x}}_i - \bar{\mathbf{x}})^T$$

Onde cada objeto denotado pelo subscrito i é a aplicação daquele operador na partição $\mathbf{X}_i$. Levando esta lógica um pouco a diante, pode-se obter uma aproximação de $\mathbf{S}$ denominada $\widetilde{\mathbf{S}}$ através de

$$n\widetilde{\mathbf{S}} = \sum_{i=1}^{s} \widetilde{\mathbf{V}}_i \widetilde{\mathbf{\Lambda}}_i^2 \widetilde{\mathbf{V}}_i^T + \sum_{i=1}^{s} n_i (\bar{\mathbf{x}}_i - \bar{\mathbf{x}})(\bar{\mathbf{x}}_i - \bar{\mathbf{x}})^T$$

Esta equação é a base do algoritmo distribuído de PCA. Nele, o lado direito será calculado paralelamente para cada partição $\mathbf{X}_i$ e depois $\widetilde{\mathbf{S}}$ será calculada no núcleo central.

Desenvolvido por Qu *et al.* (2002) e levemente adaptado para este trabalho, o algoritmo de PCA distribuída é o seguinte:

Algoritmo 1: PCA Distribuída

Entrada: $\mathbf{X}, k, s$
Saída: Os k primeiros componentes principais de $\mathbf{X}$

1 Quebrar $\mathbf{X}$ em s partições
2 **para** $i \in 1..s$ **faça**
3 Enviar $\mathbf{X}_i$ e k para o núcleo ou trabalhador C_i
4 **fim**
5 **para** *cada núcleo* C_i **faça**
6 Calcular estatísticas descritivas $(n_i, \bar{\mathbf{x}}_i, \widetilde{\mathbf{V}}_i, \widetilde{\mathbf{\Lambda}}_i)$
7 Calcular $\widetilde{\mathbf{S}}_i = \widetilde{\mathbf{V}}_i \widetilde{\mathbf{\Lambda}}_i^2 \widetilde{\mathbf{V}}_i^T + n_i (\bar{\mathbf{x}}_i - \bar{\mathbf{x}})(\bar{\mathbf{x}}_i - \bar{\mathbf{x}})^T$
8 Transmitir $\widetilde{\mathbf{S}}_i$ para localização central
9 **fim**
10 Calcular $n\widetilde{\mathbf{S}} = \sum_{i=1}^{s} \widetilde{\mathbf{S}}_i$
11 Realizar a decomposição espectral de $n\widetilde{\mathbf{S}}$ em $\mathbf{V}\mathbf{\Lambda}^2\mathbf{V}^T$
12 Selecionar os primeiros k componentes de $\mathbf{V}$ em $\hat{\mathbf{V}}$
13 **retorna** $\hat{\mathbf{Z}} = (\mathbf{I}_n - n^{-1}\mathbf{1}\mathbf{1}^T)\mathbf{X}\hat{\mathbf{V}}^T$

Em virtude da natureza teórica do algoritmo 1, antes de apresentar a sua implementação prática, algumas observações se fazem necessárias:

- Apesar de s ser considerada uma entrada do programa, é essencial que as funções distribuídas do `ddR` possuam o mesmo protótipo que suas equivalentes sequenciais, sendo necessário portanto que s seja determinada internamente;
- Pela forma como a `dmapply()` funciona, não é necessário fazer o laço da linha 2 separadamente do da linha 5 e
- Como a PCA Distribuída foi desenvolvida para bases de dados massivas e distribuídas, $\widetilde{\mathbf{S}}$ é uma aproximação para $\mathbf{S}$, fazendo com que $\hat{\mathbf{Z}}$ também haja de ser uma aproximação para os primeiros k componentes principais de $\mathbf{X}$.

Abaixo é possível ver a implementação da função `do_eq_10()`, que implementa as operações do laço da linha 5. O código que realiza estas transformações é bastante direto, sem ter necessitado de nenhuma adaptação.

```r
# Calculate equation 10
do_eq_10 <- function(X_i, X_bar, k_i) {

  # Basic params
  n_i <- nrow(X_i)

  # Calculate X_i's column means
  X_bar_i <- matrix(colMeans(X_i), ncol = 1)

  # Run SVD
  X_svd <- svd(X_i, nu = 0, nv = ncol(X_i))
  V_i <- X_svd$v
  Lambda_i <- diag(X_svd$d)

  # Tilde (select cols)
  Lambda_tilde_i <- Lambda_i[,1:k_i]
  V_tilde_i <- V_i[,1:k_i]

  # Run both parts of equation 10
  a <- V_tilde_i %*% (Lambda_tilde_i)**2 %*% t(V_tilde_i)
  b <- n_i * (X_bar_i - X_bar) %*% t(X_bar_i - X_bar)
  out <- a + b


  return(out)
}
```

Código 3.1: *Código que realiza a aplicação da linha 5 do Algoritmo 1. Seu nome faz referência à equação número 10 do artigo original no qual o algoritmo se inspira.*

Diferentemente da função `do_eq_10()`, o resto do código deve ser executado sequencialmente na máquina central do *cluster* (se o código estiver rodando em modo distribuído).

Abaixo é possível ver a parte mais essencial da função `dprcomp()`[1], onde são realizadas as contas principais do algoritmo. O resto do código foi omitido porque ele, em sua maioria, com transformações de entradas e saídas.

```r
# Distributed implementation of stats::prcomp
dprcomp <- function(x, retx = TRUE, center = TRUE,
                    scale. = FALSE, tol = NULL, rank. = NULL) {

  # [...]

  # Map over calculation of equation 10
  eq_10 <- ddR::dmapply(do_eq_10, parts(X), MoreArgs = list(X_bar = X_bar, k_i
      = rank.))

  # Reduce by adding
  n_S_tilde <- Reduce('+', ddR::collect(eq_10))

  # Calculate global PCs
```

[1]Seguindo o padrão do ddR de apenas adicionar um "d"antes do nome das funções, a aplicação de PCA distribuída se chama dprcomp() porque a sua equivalente sequencial se chama prcomp()

```
40    n_S_tilde_eigen <- eigen(n_S_tilde)
41    V <- n_S_tilde_eigen$vectors
42    Lambda <- n_S_tilde_eigen$values
43
44    # Filter columns
45    Lambda_hat <- sqrt(Lambda[1:rank.]/nrow(X))
46    V_hat <- V[,1:rank.]
47
48    # [...]
49
50    # Create Z_hat with column-centered X
51    Z_hat <- (X_cc %*% V)[,1:rank.]
52
53    # [...]
54  }
```

Código 3.2: *Excerto do código que aplica a parte central do algoritmo 1.*

Apesar de isso acabar desacelerando um pouco o desempenho geral do algoritmo, a `dprcomp()` não só implementa exatamente o mesmo protótipo que a `prcomp()`, mas também implementa exatamente a mesma saída. O objeto retornado pela PCA distribuída tem os mesmos componentes, atributos e formato que o retorno da sua equivalente sequencial, permitindo que código já existente seja reutilizado com o mínimo de modificações possível.

Fizeram-se necessárias apenas duas alterações estruturais no algoritmo de PCA. A primeira delas é o descarte do parâmetro `tol` na versão distribuída, porque seria impossível inserir o conceito de tolerância na aproximação $\widetilde{\mathbf{S}}$. E segundo lugar, na `prcomp()`, o argumento `center` pode assumir apenas dois valores: `TRUE` e `FALSE` (indicando se as colunas da tabela `X` devem ou não ser centralizadas antes da aplicação da PCA); na `dprcomp()`, se esta informação estiver disponível, o usuário pode passar diretamente para `center` as médias das colunas `X` de modo a acelerar o cálculo de `X_bar`[2]

Por fim, os testes unitários desenvolvidos para garantir a assertividade do algoritmo foram baseados na própria documentação da função `prcomp()`; lá é possível encontrar um *script* que gera tabelas artificiais perfeitas para uma tarefa de análise de componentes principais. Os testes usam essas tabelas para conferir, principalmente, o quão próximos são os resultados gerados pela `dprcomp()` dos padrões da `prcomp()`. A medida de proximidade escolhida foi a distância euclidiana entre cada um dos vetores de componentes principais e certifica-se que a média de todas as distâncias não passa de 0,05. Foi verificado experimentalmente que conforme aumenta o tamanho da tabela de entrada, melhor a aproximação de $\widetilde{\mathbf{S}}$.

3.1.2 Dboot

Bootstrap é um algoritmo de reamostragem desenvolvido por Efron (1979). A teoria matemática formal que justifica a efetividade deste método está além do escopo deste trabalho, pois envolve conceitos extremamente avançados de Estatística e o algoritmo na prática é intuitivo e facilmente compreensível sem a justificativa teórica por trás.

O *bootstrap*[3] permite obter uma medida de acurácia (definida em termos de viés, variân-

[2] Argumentos que podem receber múltiplos tipos de objetos são bastante comuns no R porque os seus programadores já estão bastante acostumados com o fato de ela ser uma linguagem fracamente tipada

[3] O termo "bootstrap"em inglês significa algo como "alça das botas". O uso desta palavra para descrever o método provém da expressão anglófona "to pull oneself up by the bootstraps", significando algo como "aproveitar ao máximo os seus poucos recursos".

cia, intervalo de confiança, erro de predição e assim por diante) para estimativas amostrais. Essa técnica permite a estimação da distribuição amostral de praticamente qualquer estatística por meio de métodos aleatórios de amostragem.

Este processo é, mais tecnicamente, a prática de estimar propriedades de estimadores medindo estas propriedades quando retirando amostras de uma distribuição aproximada. Uma escolha padrão para a distribuição aproximada é a função da distribuição empírica dos dados observados. No caso onde pode-se dizer que um conjunto de observações é proveniente de uma população independente e identicamente distribuída, o método pode ser implementado com a construção de diversas reamostragens com reposição dos dados observados (e de tamanho igual ao conjunto de dados observado).

O caso mais básico de *bootstrap* pode ser ilustrado a partir de uma amostra aleatória de tamanho n observada de uma distribuição de probabilidade completamente desconhecida F.

$$X_i = x_i, \quad X_i \sim_{\text{ind}} F$$

Com $i = 1, 2, \ldots, n$. Seja também $\mathbf{X} = (X_1, X_2, \ldots, X_n)$ uma amostra aleatória e $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ a sua realização. No *bootstrap* constrói-se uma distribuição de probabilidade amostral $\hat{F}$ colocando massa n^{-1} em cada ponto $x_1, x_2, \ldots, x_n$. Com $\hat{F}$ fixado, tira-se uma amostra de tamanho n de $\hat{F}$:

$$X_i^* = x_i^*, \quad X_i^* \sim_{\text{ind}} \hat{F}$$

Denominando $\mathbf{X}^* = (X_1^*, X_2^*, \ldots, X_n^*)$ e $\mathbf{x}^* = (x_1^*, x_2^*, \ldots, x_n^*)$ como amostra de *bootstrap*. Note que esta não é uma distribuição de permuta porque os valores de $\mathbf{X}^*$ são selecionados de $x_1, x_2, \ldots, x_n$ com reposição. O ponto chave do algoritmo é aproximar a distribuição amostral da variável aleatória $R(\mathbf{X}, F)$ através da distribuição de *bootstrap* de

$$R^* = R(\mathbf{X}^*, \hat{F})$$

Ou seja, a distribuição de R^* induzida a partir do mecanismo aleatório descrito na penúltima equação, com $\hat{F}$ fixado em seu valor observado.

Sem entrar em detalhes técnicos, quanto mais reamostragens forem realizadas, melhor o poder de aproximação da técnica de *bootstrap*. A necessidade de melhores aproximações, no entanto, transformam este método em um processo computacionalmente muito intenso e que pode se beneficiar enormemente de processamento paralelo.

A implementação distribuída do *bootstrap* apresentada aqui se baseia fortemente no pacote `boot` desenvolvido por Canty e Ripley (2017) e na teoria de Davison e Hinkley (1997). A função `boot()` já possuía um esqueleto de paralelismo que visava ajudar o usuário a melhorar o seu desempenho, logo os autores se aproveitaram desta característica e substituíram as primitivas de paralelismo encontradas na `boot()` pelas do pacote `ddR`, criando a função `dboot()`.

Grande parte da `dboot()` é voltada somente para o pré-tratamento dos dados e parâmetros de modo a realizar a reamostragem de forma correta. Abaixo é possível ver a parte central do código da `dboot()`, dedicada a criar a função de cálculo e paralelizar a sua aplicação:

```
1 # Distributed implementation of boot::boot()
2 dboot <- function (
3   data, statistic, R, sim = "ordinary", stype = c("i", "f", "w"),
```

```
  strata = rep(1, n), L = NULL, m = 0, weights = NULL,
  ran.gen = function(d, p) d, mle = NULL, simple = FALSE, ...) {

  # [...]

  # Function generator for bootstrap application
  pred.i <- NULL
  fn <- if (sim == "parametric") {
    # [...]
  }
  else {

    # [...]

    if (stype %in% c("f", "w")) {

      f <- boot::freq.array(i)
      rm(i)
      if (stype == "w") {
        f <- f/ns
      }

      if (sum(m) == 0L) {
        function(r) statistic(data, f[r, ], ...)
      } else {
        function(r) statistic(data, f[r, ], pred.i[r,], ...)
      }
    } else if (sum(m) > 0L) {
      function(r) statistic(data, i[r, ], pred.i[r, ], ...)
    } else if (simple) {
      function(r) statistic(data, boot:::index.array(n, 1, sim, strata,
        m, L, weights), ...)
    } else {
      function(r) statistic(data, i[r, ], ...)
    }
  }

  # Run bootstrap
  RR <- sum(R)
  res <- ddR::collect(ddR::dlapply(seq_len(RR), fn))

  # [...]
}
```

Código 3.3: *Excerto da função dboot() que cria a função de cálculo da estatística e paraleliza o processo de reamostragem.*

O problema de paralelizar as reamostragens é um problema vergonhosamente paralelizável, mas adaptar os parâmetros da função `boot()` foi a tarefa mais complexa:

- O esqueleto de paralelismo da função original dependia de três argumentos extras e de um processo considerável de montagem do *framework*, mas tudo isso teria que ser removido;
- Os argumentos de paralelismo tiveram que ser retirados, pois com o `ddR` o usuário pode controlar estas características globalmente no código e

- O processo de montagem do *framework* foi substituído pelo processo interno da `dmapply()`, permitindo que o código também possa ser executado no contexto distribuído.

No final, a função `dboot()` mantém a grande maioria das características da `boot()` (com a exceção do antigo processo de paralelização). Inclusive, por causa da manutenção da compatibilidade, todas as saídas da `dboot()` funcionam perfeitamente com as outras funções do pacote `boot` que servem para auxiliar no processo de *bootstrap*, incluindo as funções que geram medidas resumo e gráficos a partir de um objeto retornado pela `boot()`.

Diferentemente da `dprcomp()`, a `dboot` não é uma aproximação do algoritmo original de *bootstrap* mas sim uma paralelização sem comprometimentos do método. Como o processo de *bootstrap* envolve um elemento aleatório na reamostragem, pode ser que os resultados de duas execuções consecutivas sejam ligeiramente diferentes, mas isso não se dá em virtude de algum erro no código.

A função `dboot()` também recebeu um conjunto de testes unitários para que fosse garantida a sua assertividade durante o desenvolvimento. Estes testes usam tabelas e funções extraídas de um tutorial *web* (Kabacoff, 2017) sobre *bootstrapping*.

Capítulo 4

Desempenho

Aqui são apresentados os testes de desempenho realizados sobre as funções desenvolvidas no capítulo 3. A primeira seção deste capítulo aborda a metodologia dos testes e, mais tarde, são apresentados e discutidos os resultados obtidos.

4.1 Metodologia

Para avaliar o desempenho dos algoritmos desenvolvidos, foram utilizadas 4 máquinas virtuais limpas obtidas no Google Cloud Platform (GCP). Todas as máquinas possuíam 4 vCPUs e 15 GB de memória RAM sobre os quais foi instalada a versão 16.04 LTS do sistema operacional Ubuntu e a versão 3.2 do `R`. Para os testes *multicore* foi utilizada apenas uma das máquinas (sempre a mesma, denominada "mestre") e para os testes distribuídos foram utilizadas as 4 máquinas em um cluster SSH construído sobre a própria rede interna da GCP.

Os testes *multicore* utilizavam o *backend* `fork` do `ddR`. Esta infraestrutura usa o comando `fork` do Linux para lançar cada computação em um diferente fluxo de execução, sendo que o único parâmetro possível para este *backend* é o número máximo de fluxos rodando simultaneamente.

Já os testes distribuídos utilizavam o *backend* `parallel` do `ddR`. Esta infraestrutura usava um *cluster PSOCK* para conectar todas as 4 máquinas via SSH e rodar cada fluxo de execução em alguma das 3 máquinas denominadas "servas". Neste caso, era possível determinar quantos fluxos simultâneos poderiam ser executados em cada máquina serva e quantas máquinas deveriam ser utilizadas; para garantir equilibro, sempre que possível todas as servas eram inicializadas com o mesmo número de fluxos.

Todas as funções executadas durante os testes foram executadas 5 vezes com o pacote `microbenchmark` (Mersmann, 2018) para garantir a captura da variabilidade natural dos tempos de execução. Nos gráficos são apresentadas as medianas nos tempos de execução como métrica principal e, através de barras de erro, são destacados os primeiros e terceiros quartis de cada experimento.

Para informações mais detalhadas sobre como foram realizados os testes, ela é oferecida no apêndice A.

4.2 Dprcomp

Abaixo é possível ver os resultados dos testes da `dprcomp()` contra a `prcomp`. Os testes de desempenho envolveram a extração dos componentes principais de um conjunto de dados artificial de 240 mil linhas e 70 colunas (uma base considerada de mediana para grande

neste tipo de algoritmo). Especificamente os testes da PCA distribuída com a `dprcomp()` tiraram vantagem da nova funcionalidade do argumento `dprcomp()` de receber de antemão as médias das colunas (como descrito nas seções anteriores).

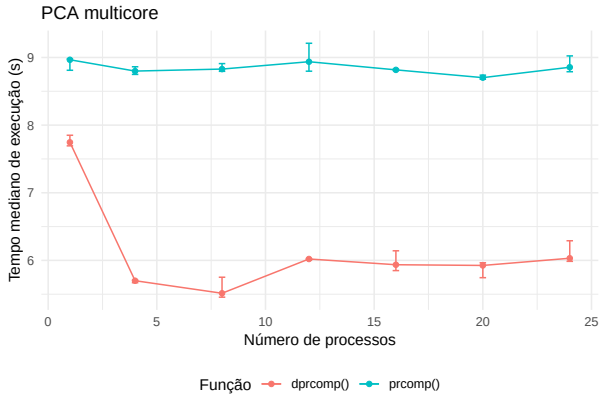

Figura 4.1: *PCA multicore.*

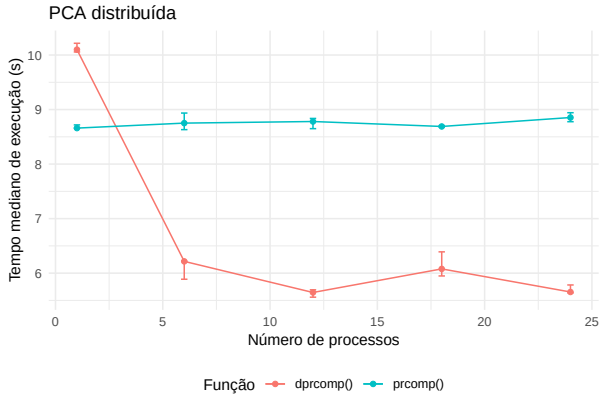

Figura 4.2: *PCA distribuída.*

Como é possível observar nas figuras 4.1 e 4.2, a `dprcomp()` supera o algoritmo sequencial em desempenho em todos os casos de teste exceto quando executada em apenas um fluxo remotamente. Considerando que (pelo menos no caso *multicore*) não houve absolutamente nenhuma diferença entre as entradas da `dprcomp()` e da `prcomp()`, é possível dizer que o ganho de desempenho é considerável.

Apesar de não ser possível encontrar uma causa formal para a semelhança do desempenho da `dprcomp()` em ambos os cenários de paralelismo, os autores consideram altamente provável que o *overhead* de transferir tantos dados para o servos remotos possa estar impondo um limite no ganho de desempenho do algoritmo quando ele é executado no modo distribuído.

4.3 Dboot

Já os resultados dos testes da `dboot()` contra a `boot()` contam uma história completamente diferente. Estes testes envolveram o cálculo do intervalo de confiança 95% para os três coeficientes de um modelo de regressão aplicado a uma base bastante simples (32 linhas e 11 colunas). As entradas para ambas as funções foram iguais em todos os casos de teste, então os gráficos representam exatamente os mesmos cálculos.

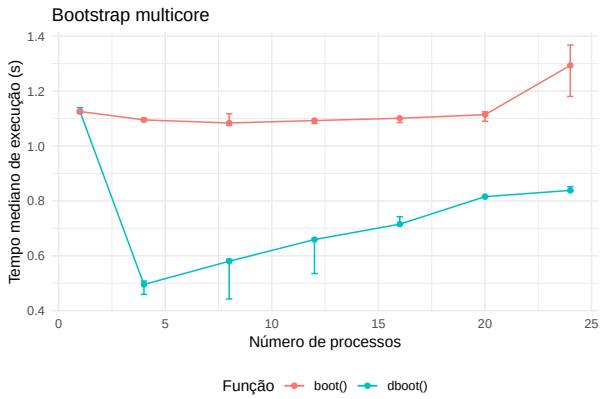

Figura 4.3: *Bootstrap multicore.*

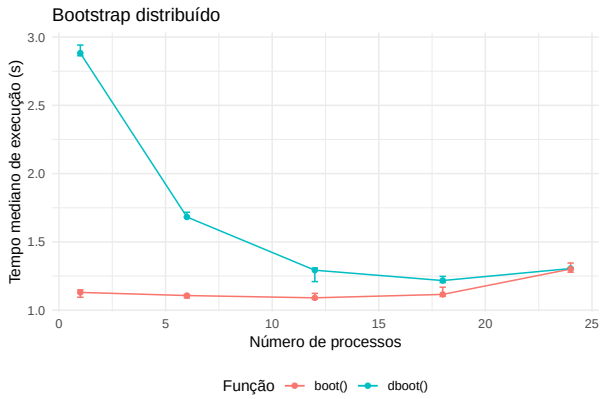

Figura 4.4: *Bootstrap distribuída.*

Diferentemente da `dboot()`, como é possível observar nas figuras 4.3 e 4.4, somente o algoritmo de *bootstrap* no contexto *multicore* ganha consistentemente da `boot()` (mesmo havendo saturação dos ganhos depois dos 4 fluxos de execução).

Isso parece ser um indicativo de que a tarefa tarefa era pequena demais para tirar vantagem completa do processamento distribuído (indicado pelos tempos de execução no geral muito pequenos), mas, exceto por isso, é possível hipotetizar que o algoritmo de *bootstrap* escala em desempenho muito menos do que o de PCA.

Capítulo 5

Conclusões

Neste capítulo é realizado um breve fechamento do trabalho, retomando os objetivos do mesmo e se estes foram alcançados. Também são disponibilizados os links para todos os códigos utilizados ao longo do texto e também os dados resultantes dos testes de performance. Por fim, são destacados alguns possíveis próximos passos para a extensão do `ddR`.

5.1 Fechamento

Os trabalhos deste texto começaram a ser realizados entre o final de março e o começo de abril de 2018. No primeiro semestre foi escrita a grande maioria do código, mas alguns ajustes finais (principalmente relacionados ao desempenho e documentação dos programas) foram feitos juntamente com a escrita da monografia.

O texto em si foi escrito a partir de agosto de 2018, sendo a maior parte dele concluída após a realização dos testes de desempenho em novembro do mesmo ano. Algumas modificações estruturais foram feitas na semana anterior à sua entrega atendendo aos pedidos do orientador do trabalho, mas seu conteúdo permaneceu praticamente inalterado.

O maior desafio do projeto todo girou em torno do algoritmo de PCA distribuída. Os autores do artigo no qual a implementação baseou-se disponibilizaram diversas versões preliminares do algoritmo e, mesmo na versão final, ainda possuía alguns erros de notação. Por isso, quase dois meses foram necessários até que os autores conseguissem obter uma versão funcional e assertiva da `dprcomp()`. Nesta monografia, todas as equações referentes a este algoritmo estão corrigidas e correspondem exatamente ao código da função implementada.

Outra dificuldade importante encontrada ao longo do caminho foi a montagem do *cluster* para realização dos testes distribuídos. Os autores do `ddR` escreveram também uma programoteca dedicada para a montagem de *clusters* denominada `distributedR.ddR`, mas que foi abandonada há 3 anos e possuía diversos problemas de compatibilidade; vários dias foram gastos tentando utilizar e modernizar esta biblioteca até que os autores decidiram tomar um caminho alternativo e usar os *clusters PSOCK* suportados pelo *backend* `parallel` do `ddR`.

Alguns detalhes extras sobre a experiência dos autores com programação em `R` e algoritmos paralelos são oferecidos no B.

5.1.1 Motivação e Objetivo

A motivação pessoal dos autores por trás deste trabalho era se aprofundar tanto em programação paralela e distribuída quanto na programação em `R`; no primeiro tema por causa da importância de aplicações de alto desempenho no mundo moderno e, no segundo,

por causa do crescente uso da linguagem em aplicações cada vez mais científicas e relevantes (aprendizado de máquina, *big data*, etc.).

A extensão do `ddR` foi o projeto escolhido porque ele apresentava uma intersecção interessante e inusitada dos dois temas destacados acima. Desenvolvido por membros da HP Vertica, este *framework* se propunha a conectar diversas tecnologias de concorrência e distribuição sob uma única interface, facilitando o desenvolvimento por parte de programadores que, muitas vezes, não têm formação em áreas da Computação. Além disso, a proposta do pacote de disponibilizar algoritmos de aprendizado de máquina paralelizados mas com os mesmos protótipos e saídas das suas versões sequenciais era muito promissora.

O objetivo deste trabalho foi, portanto, adicionar novos algoritmos ao repertório do `ddR` para que os usuários tivessem ainda mais funcionalidades disponíveis ao usar este *framework*. Após alguma consideração foi determinado que os melhores candidatos a implementação eram os algoritmos de análise de componentes principais e de *bootstrap*.

Apesar de alguns contratempos no desenvolvimento e avaliação de desempenho, dados os resultados alcançados os autores consideram que os abjetivos determinados foram cumpridos. Ambos os algoritmos foram implementados (com comentários, documentação e testes unitários) e publicados em bibliotecas individuais que podem ser facilmente instaladas por qualquer usuário de `R`. Além disso, o ganho de desempenho das funções paralelas foi testado e comprovado, permitindo que um usuário já possa utilizá-las em qualquer aplicação e obter pelo menos algum benefício marginal praticamente sem ônus.

5.1.2 Códigos e Avaliação de Desempenho

Esta monografia, juntamente com todos os arquivos relevantes para si, está disponível publicamente no repositório https://github.com/clente/tcc. Os arquivos LaTeX que formam este trabalho estão disponíveis em https://github.com/clente/tcc/tree/master/Thesis/LaTeX, já o PDF completo resultante da compilação está em https://github.com/clente/tcc/blob/master/Thesis/LaTeX/monografia_template.pdf. O código utilizado para realização dos testes (e seus resultados) estão em https://github.com/clente/tcc/blob/master/Thesis/Code/benchmark.R, enquanto os código para a geração dos gráficos está em https://github.com/clente/tcc/blob/master/Thesis/Code/plots.R. O arquivo CSV que contém exclusivamente os resultados dos testes de desempenho pode ser encontrado no mesmo repositório em https://github.com/clente/tcc/blob/master/Thesis/Code/benchmark.csv.

A versão do `ddR` sobre a qual foram feitas as extensões deste trabalho se encontra em https://github.com/clente/ddR. A biblioteca com a `dprcomp()` está em https://github.com/clente/ddR/tree/master/algorithms/prcomp.ddR e o arquivo com seu código completo está em https://github.com/clente/ddR/blob/master/algorithms/prcomp.ddR/R/dprcomp.R. Já a biblioteca com a `dboot()` está em https://github.com/clente/ddR/tree/master/algorithms/boot.ddR e o arquivo com o seu código está em https://github.com/clente/ddR/blob/master/algorithms/boot.ddR/R/dboot.R.

Para mais informações sobre este texto, a proposta de trabalho e o pôster apresentado no Instituto de Matemática e Estatística da Universidade de São Paulo, a página permanente dele está em https://linux.ime.usp.br/~clente/mac0499/.

5.2 Trabalhos futuros

Atualmente os autores enxergam três caminhos possíveis para continuar os trabalhos aqui descritos. O primeiro seria adicionar mais algoritmos para o `ddR`, o segundo envolveria a

criação de novos *backends* com os quais o `ddR` poderia se conectar e o terceiro seria melhorar o código da interface da `dmapply()`.

Como já foi descrito anteriormente, a área de aprendizado de máquina tem avançado a passos largos e velozes; novos algoritmos são desenvolvidos praticamente todo dia e os usuários estão cada vez mais conectados com estas tecnologias. Isso quer dizer que ainda há espaço para a integração de diversas outras funções ao repertório do `ddR`, incluindo máquinas de suporte vetorial, modelos aditivos generalizados, métodos mais avançados de *bootstrap*, dentre muitos outros. Como a interface do `ddR` permite a instalação de novos módulos separados do pacote principal, diversas pessoas poderiam participar destas extensões sem comprometer a sincronização ou qualidade do código já existente.

A possibilidade de aumentar o número de *backends* já é uma tarefa consideravelmente mais complexa. Os autores do `ddR` disponibilizaram pelo menos 3 *backends* completos (`fork`, `parallel` e `DistributedR`) e um *backend* parcialmente implementado (`SparkR`). Outras tecnologias de programação concorrente e distribuída como OpenMP, MPI, etc., poderiam ser implementadas de modo que o `ddR` fosse ainda mais flexível e oferecesse uma barreira ainda menor a um usuário comum. O maior problema com esta extensão, no entanto, é que os *backends* precisam de uma integração muito forte com o código do `ddR` e portanto pode ser extremamente difícil implementá-los sem o auxílio dos autores originais.

O terceiro caminho para melhorar o `ddR` é o mais difícil. Os autores deste trabalho fizeram alguns testes e identificaram gargalos enormes na implementação das estruturas de dados distribuídas oferecidas pelo pacote, mas para melhorar o seu desempenho seria necessário autorização completa dos desenvolvedores originais para modificar o seu código fonte. Além disso, ao longo do ano de 2018, o `ddR` foi declarado como abandonado pelos seus autores e o desenvolvimento de novas funcionalidades foi interrompido; isso significa que é improvável que novas adições sejam feitas ao pacote num futuro próximo.

5.3 Agradecimentos

Por este projeto, agradeço primeiramente ao meu orientador, o prof. dr. Alfredo Goldman, e à coordenadora da disciplina MAC0499, a profa. dra. Nina Hirata. Agradeço também a Edward Ma, Indrajit Roy e Michael Lawrence, desenvolvedores originais do `ddR`, por terem respondido meus e-mails com dúvidas acerca do pacote e por terem encorajado o meu trabalho.

Apêndice A

Testes de Desempenho

Como já foi dito anteriormente, os testes de desempenho da funções desenvolvidas foram realizados em um ambiente isolado e com o mínimo de programas instalados em cada máquina (apenas o mínimo necessário para rodar os algoritmos e conectar as máquinas via SSH).

No total foram utilizadas 4 máquinas virtuais do Google Cloud Platform (GCP) com 4 vCPUs, 15 GB de memória RAM e um disco de 30GB. Nestas MVs foi instalado o Ubuntu 16.04 LTS e o `R` versão 3.2. Cada vez que um novo teste era executado, a sessão `R` era reiniciada e o teste era replicado 5 vezes na mesma sessão para capturar a variabilidade natural das execuções. O controle do número de fluxos simultâneos (tanto no contexto *multicore* quando no distribuído) era realizado diretamente pelas diretivas do `ddR`.

A.1 Código

Para os testes da `dprcomp()`, foi utilizada uma função de geração de dados descrita na documentação da `prcomp()`. Depois que tivessem sido criados os dados, os parâmetros de paralelismo eram passados para o `ddR` e, finalmente, os testes de desempenho eram executados. Abaixo é possível ver exatamente os códigos utilizados para o *benchmark*.

```
# Carregar prcomp.ddR e ddR
pkgload::load_all()
library(ddR)
library(microbenchmark)

# Gerar dados
generate_data <- function(n_row, n_col) {

  # Create sample data
  C <- chol(S <- toeplitz(.9 ^ (0:(n_col-1))))
  M <- matrix(rnorm(n_row*n_col), n_row, n_col)
  X <- M %*% C

  return(X)
}

# Gerar dados
Y <- generate_data(240000, 70)

```

```
# Executar em multicore com nExec fluxos
useBackend("fork", executors = nExec)

# Rodar teste de desempenho
microbenchmark(dprcomp(X), prcomp(Y), times = 5)

# Gerar dados para contexto distribuído
Y <- generate_data(240000, 70)
X_bar <- colMeans(Y)
X <- as.darray(Y, c(10000, 70))

# Executar distribuído com nExec fluxos em cada máquina
useBackend("parallel", c(rep(ip1, nExec), rep(ip2, nExec), rep(ip3, nExec)))

# Rodar teste de desempenho
microbenchmark(dprcomp(X, center = X_bar), prcomp(Y), times = 5)
```

Código A.1: *Código que mede o desempenho da dprcomp() em comparação com a prcomp().*

Já para os testes da `dboot()`, foi utilizada uma função de estatística descrita em um tutorial de *bootstrap* e dados que já vêm embutidos no R. Depois que parâmetros de paralelismo eram passados para o `ddR` os testes de desempenho eram executados. Abaixo é possível ver exatamente os códigos utilizados para este *benchmark*.

```
# Carregar boot.ddR e ddR
pkgload::load_all()
library(ddR)
library(microbenchmark)

# Estatística a ser medida
bs <- function(formula, data, indices) {
  d <- data[indices,]
  fit <- lm(formula, data=d)
  return(coef(fit))
}

# Executar em multicore com nExec fluxos
useBackend("fork", executors = nExec)

# Rodar teste de desempenho
microbenchmark(
  dboot = dboot(mtcars, bs, R=1000, formula=mpg~wt+disp),
  boot = boot::boot(mtcars, bs, R=1000, formula=mpg~wt+disp),
  times = 5)

# Executar distribuído com nExec fluxos em cada máquina
useBackend("parallel", c(rep(ip1, nExec), rep(ip2, nExec), rep(ip3, nExec)))

# Rodar teste de desempenho
microbenchmark(
  dboot = dboot(mtcars, bs, R=1000, formula=mpg~wt+disp),
  boot = boot::boot(mtcars, bs, R=1000, formula=mpg~wt+disp),
  times = 5)
```

Código A.2: *Código que mede o desempenho da dboot() em comparação com a boot().*

Apêndice B

R na Prática

Os autores deste trabalho possuem anos de experiência profissional e acadêmica com a linguagem `R` de programação, o que contribuiu muito para a escolha do tema do projeto como um todo. Um dos autores conheceu o `ddR` ao procurar soluções eficientes para problemas de desempenho que ele estava encontrando no seu trabalho; o mesmo também escreve frequentemente sobre ciência de dados em `R` na sua página pessoal e também dá aulas particulares dentro e fora da universidade sobre a linguagem.

Justamente por ter um contato muito grande com a linguagem, os autores conhecem o seu grande potencial justamente no tocante a ciência de dados. Com o `R` é possível aplicar modelos estatísticos extremamente complexos, executar tarefas de *deep learning*, criar *dashboards* extremamente elegantes e interativos com apenas algumas linhas de código, gerar relatórios e apresentações automatizados utilizando *frameworks* como `Beamer`, `Pandoc` e outros, etc.

Apesar de uma certa resistência por parte da comunidade da computação de adotar o `R` como principal ferramenta de análise de dados (geralmente optando pelo `Python`), é importante ressaltar que, diferentemente das outras linguagens o `R` foi criada pensando em dados como o seu objeto central. Essa decisão permite a criação de *pipelines* enxutas de tratamento de dados e análises reprodutíveis (Xie, 2017) sem praticamente nenhum ônus sobre o programador.

B.1 Caso de Uso

Para ilustrar o potencial da linguagem, os autores ressaltam um projeto desenvolvidos por eles no final de 2018. A pesquisadora Fernanda Estevan apresentou aos autores um código de modelagem extremamente complexo, mas que não possuía o desempenho desejado.

Apesar de conter operações bastante custosas com os dados (produtos matriciais, inversões matriciais e assim por diante), o verdadeiro problema se encontrava em um laço que servia para numericamente se aproximar do resultado desejado. Com poucas adaptações e uma primitiva de paralelismo o código já executava 30 vezes mais rápido.

Ao longo do mês de dezembro os autores pretendem também integrar o `ddR` ao projeto de modo a acelerar ainda mais o código. Isso servirá como prova conceitual do funcionamento do pacote e dos benefícios que pode trazer aos usuários da linguagem.

Referências Bibliográficas

Canty e Ripley(2017) Angelo Canty e B. D. Ripley. *boot: Bootstrap R (S-Plus) Functions*, 2017. R package version 1.3-20. Citado na pág. 18

Davison e Hinkley(1997) A. C. Davison e D. V. Hinkley. *Bootstrap Methods and Their Applications*. Cambridge University Press, Cambridge. URL http://statwww.epfl.ch/davison/BMA/. ISBN 0-521-57391-2. Citado na pág. 18

Eddelbuettel(2018) Dirk Eddelbuettel. High-performance and parallel computing with r. https://cran.r-project.org/web/views/HighPerformanceComputing.html, 2018. Último acesso em 07/09/2018. Citado na pág. 2

Efron(1979) B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26. doi: 10.1214/aos/1176344552. URL https://doi.org/10.1214/aos/1176344552. Citado na pág. 17

Ellen e Brown(2016) Faith Ellen e Trevor Brown. Concurrent data structures. Em *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, páginas 151–153. ACM. Citado na pág. 7

Gribble *et al.*(2000) Steven Gribble, Eric Brewer, Joseph Hellerstein e David Culler. Scalable, distributed data structures for internet service construction. Em *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, página 22. USENIX Association. Citado na pág. 8

Herlihy e Shavit(2011) Maurice Herlihy e Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann. Citado na pág. 3, 10

Ihaka e Gentleman(1996) Ross Ihaka e Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314. Citado na pág. 1

Kabacoff(2017) Robert Kabacoff. Nonparametric bootstrapping. https://www.statmethods.net/advstats/bootstrapping.html, 2017. Último acesso em 30/10/2018. Citado na pág. 20

Liaw e Wiener(2002) Andy Liaw e Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22. URL https://CRAN.R-project.org/doc/Rnews/. Citado na pág. 10

Ma *et al.*(2016a) Edward Ma, Vishrut Gupta, Meichun Hsu e Indrajit Roy. Dmapply: A functional primitive to express distributed machine learning algorithms in r. *Proc. VLDB Endow.*, 9(13):1293–1304. ISSN 2150-8097. doi: 10.14778/3007263.3007268. URL https://doi.org/10.14778/3007263.3007268. Citado na pág. 10

Ma *et al.*(2016b) Edward Ma, Indrajit Roy e Michael Lawrence. *ddR: Distributed Data Structures in R*, 2016b. URL https://github.com/vertica/ddR/. R package version 0.1.3. Citado na pág. 2, 9, 10

Mersmann(2018) Olaf Mersmann. *microbenchmark: Accurate Timing Functions*, 2018. URL https://CRAN.R-project.org/package=microbenchmark. R package version 1.4-6. Citado na pág. 10, 21

Moir e Shavit(2001) Mark Moir e Nir Shavit. Concurrent data structures, 2001. Citado na pág. 7

Pearson(1901) Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572. Citado na pág. 14

Qu *et al.*(2002) Yongming Qu, George Ostrouchov, Nagiza Samatova e Al Geist. Principal component analysis for dimension reduction in massive distributed data sets. Em *Proceedings of IEEE International Conference on Data Mining (ICDM)*. Citado na pág. 15

R Core Team(2018a) R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018a. URL https://www.R-project.org/. Citado na pág. 2

R Core Team(2018b) R Core Team. R language definition. *Vienna, Austria: R Foundation for Statistical Computing*. Citado na pág. 2

TIOBE software BV(2018) TIOBE software BV. Tiobe index. https://www.tiobe.com/tiobe-index/, 2018. Último acesso em 07/09/2018. Citado na pág. 1

Venables e Ripley(2002) W. N. Venables e B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edição. URL http://www.stats.ox.ac.uk/pub/MASS4. ISBN 0-387-95457-0. Citado na pág. 10

Wickham(2011) Hadley Wickham. testthat: Get started with testing. *The R Journal*, 3 (1):5–10. Citado na pág. 14

Xie(2017) Yihui Xie. The first notebook war. https://yihui.name/en/2018/09/notebook-war/, 2017. Último acesso em 20/11/2018. Citado na pág. 33

Zaharia *et al.*(2010) Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker e Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95. Citado na pág. 8