UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JULIO COITINHO PINTO

Geração de representação distribuída de palavras fatorando a matriz de informação mútua pontual utilizando o gradiente descendente estocástico no modelo de MapReduce

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Cláudio Fernando Resin Geyer

Porto Alegre
2018

RESUMO

Este trabalho consiste na implementação de um método de fatoração de grandes matrizes de forma paralela utilizando o gradiente descendente estocástico e o adaptando ao modelo de programação MapReduce. Apresentando inicialmente ao leitor conceitos básicos de processamento de linguagem natural (PLN), da área de *BigData* e dos conceitos de *MapReduce*, este trabalho tem como objetivo aplicar técnicas atualmente utilizadas na geração de representações distribuídas de palavras ao *framework* Apache Flink, a fim de beneficiar o método com as vantagens cada vez mais presentes e acessíveis da programação distribuída e paralela. Este processo exige a manipulação de matrizes muito grandes e esparsas o que traz um grande tempo de processamento e de uso de recursos das máquinas utilizadas. A aplicação implementa todo o pré-processamento do arquivo de entrada até a etapa da aplicação do gradiente descendente estocástico, que realiza a fatoração da matriz em vetores de palavras e contextos, no modelo *MapReduce*. Com base na observação dos resultados mostramos que a implementação consegue ter um ganho de performance quando executada paralelamente, nas etapas do pré-processamento e na etapa final do modelo, que gera os vetores desejados.

Palavras-chave: Programação Distribuída e Paralela. Processamento de texto. Fatoração de Matrizes. Processamento de Linguagem Natural. BigData. Apache Flink. MapReduce.

This should be the title in English

ABSTRACT

This work consists in the implementation of a method of factoring large matrices in parallel using the stochastic descendent gradient and adapting it to the MapReduce programming model. Introducing initially the basic concepts of natural language processing (NLP), the area of BigData and the concepts of MapReduce, this work aims to apply techniques currently used in the generation of distributed representations of words to the Apache Flink *framework* in order to benefit the method with the increasingly present and accessible advantages of distributed and parallel programming. This process requires the manipulation of very large and sparse matrices which brings a great deal of time to processing and resource use of the machines used. The application implements all the pre-processing of the input file until the step of applying the stochastic descendent gradient, which performs the factorization of the matrix into vectors of words and contexts, in the model MapReduce. Based on the observation of the results we show that the implementation achieves a performance gain when executed in parallel, in the pre-processing stages and in the final stage of the model, which generates the desired vectors.

Keywords: Distributed and Parallel Programming. Text processing. Factoring of Matrices. Natural Language Processing. Big data. Apache Flink. MapReduce.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

PMI	*Point of Mutual Information*
PPMI	*Positive Point of Mutual Information*
SGD	*Stochastic Descent Gradient*
SQL	*Structured Query Language*
IBM	*International Business Machines*
DAG	*Directed Acyclic Graph*
IDC	*International Data Corporation*
API	*Application Programming Interface*
NLP	*Natural Language Processing*
PLN	Processamento de Linguagem Natural
RAM	*Random Access Memory*

SUMÁRIO

1 INTRODUÇÃO

Com o avanço cada vez maior da computação estamos produzindo uma quantidade muito grande de novos dados todos os dias. Companhias coletam trilhões de *bytes* de informações sobre seus consumidores, fornecedores e sobre suas operações, além dos vários novos sensores que estão sendo anexados no mundo físico em aparelhos como celulares e carros (MANYIKA, 2011). Toda esta transição do analógico para o digital traz consigo desafios enormes, estima-se que por dia a população mundial produza 2,5 quintilhões de bytes. Transformar toda esta informação em algo útil é um desafio que cada vez mais se mostra evidente. Do total de dados existentes no mundo hoje, 90% foi gerado nos últimos dois anos (IBM, 2017) e este crescimento não mostra sinais que irá parar ou até mesmo desacelerar.

Processar essa quantidade de dados tão grande é um problema que deve ser enfrentado e o processamento distribuído e paralelo parece ser o caminho para contornar os limites da nossa tecnologia atual. Técnicas de Big Data tem sido cada vez mais empregadas, em diversas áreas, para conseguirmos analisar e processar estes dados de forma que eles possam se tornar utilizáveis. Uma destas técnicas mais aceitas hoje é o *MapReduce*, que se propõe a diminuir a complexidade de paralelizar e distribuir tarefas enquanto foca em desempenho e tolerância a falhas (LEE, 2011). Existem diversos frameworks que aplicam o modelo de *MapReduce*, com destaque para o Apache Hadoop[1], que fornece um sistema de arquivos distribuído e funções de análise e transformações de grandes arquivos de dados, de forma confiável e escalável (SHVACHKO et al., 2010). Outros framework, mais recentes, como Apache Storm[2] , Apache Spark[3][4] e Apache Flink procuram melhorar o modelo de MapReduce, trazendo novas funcionalidades e modos de execução, para ser possível tratar inclusive grandes fluxos de dados criados em tempo de execução.

A área de Processamento de Linguagens Naturais (PLN) é uma das mais impactadas por essas mudanças da nossa tecnologia. Ela estuda meios de se interpretar e estruturar dados de linguagens naturais de forma que um computador possa nos entender (CHOWDHURY, 2003). Grande parte dos dados criados hoje são de textos não estruturados gerados na internet.

[1]Apache Hadoop, http://hadoop.apache.org/. Acessado 2 Jun. 2018.
[2]Apache Storm, http://storm.apache.org/. Acessado 2 Jun. 2018.
[3]Apache Spark, https://spark.apache.org/. Acessado 2 Jun. 2018.
[4]Apache Flink, https://flink.apache.org/. Acessado 2 Jun. 2018.

Interpretar e achar meios de utilizar essas informações é algo que muitas empresas e governos estão pesquisando. Neste trabalho é apresentado um método de paralelizar e distribuir a geração de uma matriz de representação distribuída de palavras. Para isso a matriz, de informação mútua pontual positiva (*Positive Pointwise Mutual Information* - PMI), é fatorada vía gradiente descendente estocástico (*Stochastic Gradient Descendent* - SGD) implementado no modelo de programação *MapReduce*. Apesar de o SGD ser um método que depende das informações dos passos anteriores, será implementado um modo de conseguir separar o problema em partes menores, que possam ser distribuídas por várias máquinas, para a computação poder ser realizada.

1.1 Motivação

Arquivos a serem processados estão cada vez maiores e com mais informações. Tratar estes dados, em muitas tarefas, começa a ser algo demorado e em muitos casos algo impossível utilizando apenas uma máquina. Nas tarefas que utilizam representações distribuídas de palavras, temos que gerar e manter grandes matrizes em memória para acessos rápidos. Estas matrizes têm como característica serem muito grandes e esparsas, tornando muito custoso sua criação e manutenção.

Utilizar técnicas de programação paralela e distribuída, para tentar acelerar estes problemas muito custosos, é um caminho que várias áreas estão tomando. Este trabalho tem como motivação adaptar um modelo já utilizado, de fatorar essas grandes matrizes, aos conceitos e frameworks de *BigData*.

1.2 Objetivo

O principal objetivo deste trabalho é a implementação de uma técnica de geração de representações distribuídas de palavras, fatorando a matriz de informação mútua pontual positiva (PPMI) utilizando o gradiente descendente estocástico, implementado no modelo de programação *MapReduce*. A finalidade desta implementação é a de se explorar as vantagens que o modelo *MapReduce* tem a oferecer de se distribuir tarefas para serem processadas paralelamente a fim de acelerar o processo de fatoração da matriz utilizando o gradiente descendente estocástico.

O trabalho visa avaliar se esta implementação é viável e se os resultados obtidos mostram uma melhora tanto em termos de escalabilidade da distribuição das tarefas como em termos de resultados qualitativos, comparando os resultados obtidos com o de outra solução da mesma área.

1.3 Organização do trabalho

Este texto foi organizado da seguinte forma: no segundo capítulo, são apresentados os conceitos utilizados no decorrer do trabalho e que são necessários para sua compreensão. No terceiro capítulo é apresentado o modelo proposto para o problema a ser resolvido. No quarto capítulo o protótipo é definido e suas características são detalhadas. No quinto capítulo são apresentados a metodologia utilizada, os resultados obtidos e a análise dos experimentos realizados. Por fim, no último capítulo, as conclusões do trabalho são apresentadas para, em seguida, sugerir trabalhos futuros que poderão dar continuidade a pesquisa aqui realizada.

2 CONCEITOS

Neste capítulo, serão introduzidos os conceitos que fundamentaram este trabalho. Eles servem para entender como tudo foi pensado e desenvolvido para se chegar ao resultado final. Na seção 2.1 é apresentada a área de *BigData*, os conceitos de *MapReduce* e o *framework* Apache Flink. Na seção 2.2 são descritos os conceitos de Processamento de Linguagem Natural e explorado mais a fundo a parte de vetores, palavras e matrizes de informações.

2.1 Big Data

Nas últimas décadas transformamos o modo como vivemos, estudamos e trabalhamos. Cada vez mais produzimos e consumimos dados no nosso cotidiano. O IDC's[1] estima que em 2025 iremos gerar 163 *Zetabytes* (aproximadamente um trilhão de gigabytes) por ano, dez vezes mais que no ano de 2016. Essa nova demanda, que cresce cada vez mais, trouxe os desafios de como vamos armazenar, analisar e utilizar essas informações de maneira rápida e em muitas vezes em tempo real.

Este novo paradigma da computação está sendo chamado de *BigData* e podemos explicá-lo mais claramente por um modelo frequentemente referenciado como múltiplos V's (ASSUNçãO et al., 2015), pois suas principais características e desafios começam pela letra V. Volume trata do tamanho dos dados que temos que processar. Com arquivos muito grandes, devemos pensar em formas de particionar, distribuir e reagrupar estes dados.Velocidade trata da rapidez em que os dados são criados e do tempo que levamos para trabalhá-los. Um dos campos importantes de *BigData* é o processamento em tempo real dos dados produzidos, analisando e transformando eles assim que são gerados. Variedade foca em tratar de forma fácil e eficiente os mais variados tipos de estruturas de dados, tentando oferecer soluções que não se restrinjam aos formatos estruturados. Veracidade aborda o problema da confiabilidade dos arquivos que desejamos consumir. Trabalhar com grandes conjuntos de dados, por vezes sem uma estrutura clara, traz muitas vezes dúvidas sobre a precisão e a consistência do que processamos. Em muitos casos os dados devem ser analisados e tratados antes de poderem ser utilizados. Valor refere-se à importância daquilo que conseguimos produzir transformando os dados usando *BigData*. Pensando de forma mais comercial, é o valor que uma empresa consegue obter com os resultados do seu

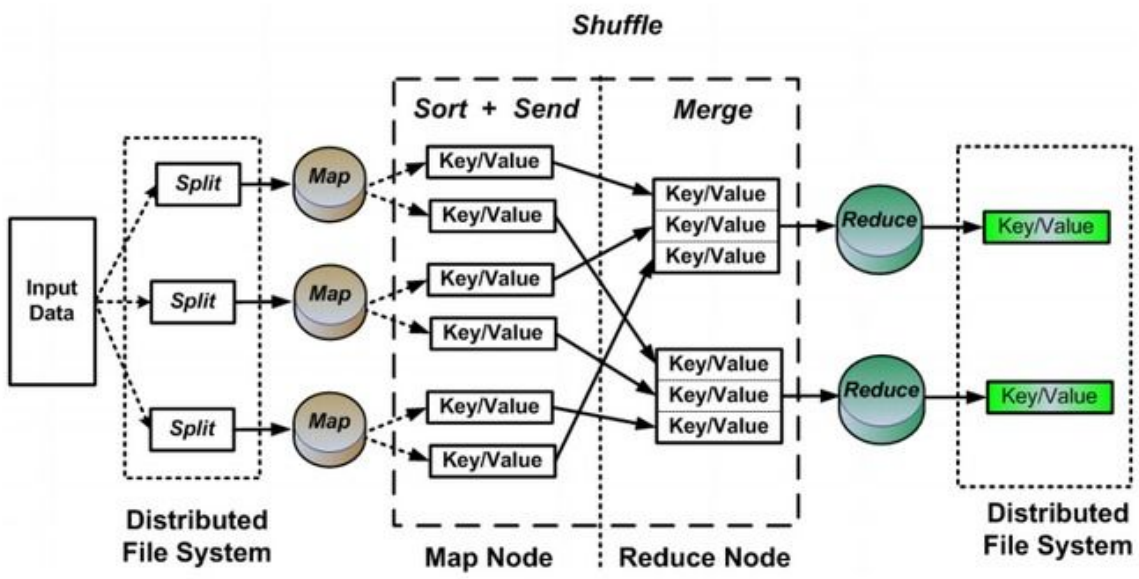
processamento dos dados. Apesar de alguns autores escolherem alguns V's diferentes para caracterizar *BigData*, variedade, velocidade e volume são tradicionalmente mencionados.

A área cresceu muito nos últimos anos e parece uma das possibilidades de como podemos acelerar nossos processamentos, utilizando nossa tecnologia atual. Se não podemos tratar de dados mais rapidamente, temos que dividir o trabalho entre várias máquinas e tratar dele paralelamente. Apesar de parecer uma saída simples para o problema, isso traz consigo vários novos desafios na programação destas aplicações, como concorrência, dependência de dados, sincronização e tolerância a falhas.

2.1.1 Map Reduce

MapReduce é um modelo de programação paralela desenvolvido pela Google que visa abstrair a complexidade de processamentos distribuídos de grandes volumes de dados. Baseado nas primitivas de *Map* e *Reduce* das linguagens funcionais, o modelo cuida de toda a parte do gerenciamento da distribuição da computação dos dados de entrada em várias máquinas (DEAN; GHEMAWAT,2008). Ao programador cabe o trabalho de desenvolver as funções de mapeamento e redução do problema.

Figura 2.1: Fluxo de Execução *MapReduce*



Fonte: Anjos et al., (2016, p.3)

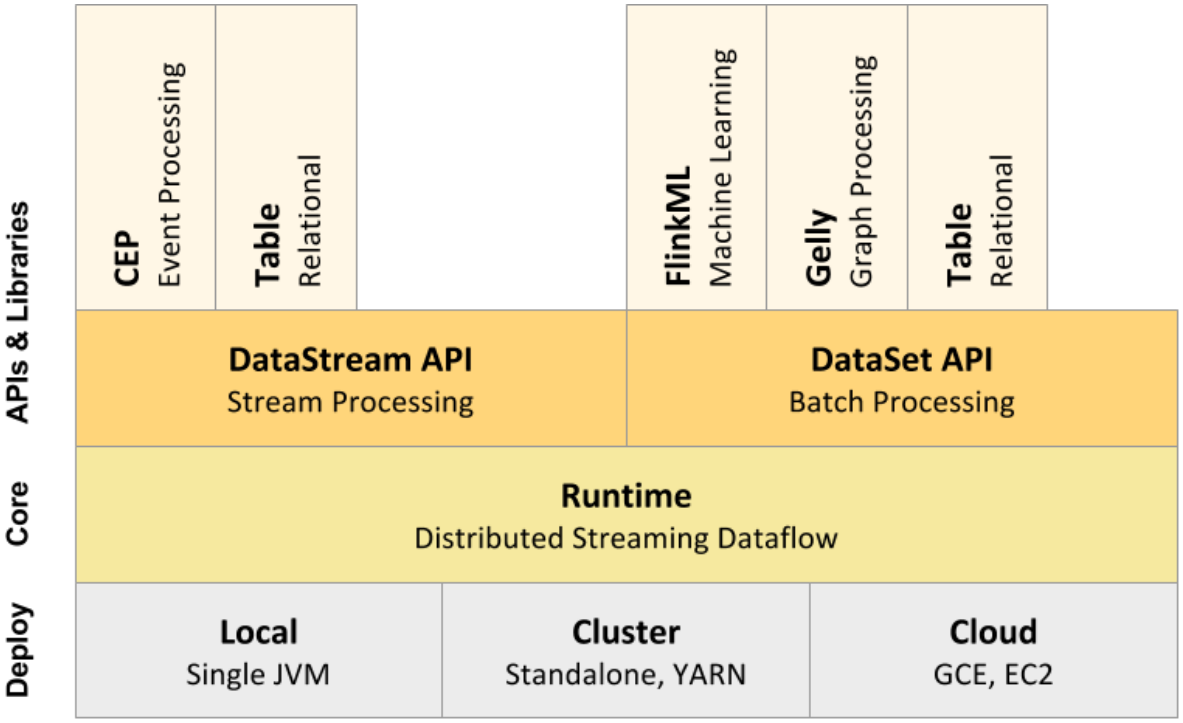
Na figura 1.1 podemos ver mais detalhadamente o fluxo de execução de uma aplicação no modelo de *MapReduce* (ANJOS et al., 2016). A aplicação replica seu código por todas as máquinas do ambiente que está rodando e uma das instâncias, chamada de *master*, é a responsável pelo gerenciamento de todo o fluxo de execução. O *master* fica responsável por além de distribuir o trabalho entre os *workers*, monitorar o progresso de cada um e agir em casos de falhas ou atrasos. As outras instâncias, chamadas de *workers*, recebem tarefas para executar do master. Os dados de entrada são divididos em pedaços menores, usualmente chamados de chunks, serializados e distribuídos pelos *workers*. Neste ponto os *workers* executam a função de mapeamento dos dados de entrada, emitindo pares de chave e valor como resultado intermediário da aplicação, estes pares ficam armazenados temporariamente na memória do próprio *worker*. Periodicamente estes pares são salvos em disco, particionados por uma função e sua localização é informada ao master, que por sua vez informa aos *workers* a localização dos dados utilizados nas funções de redução. O modelo de execução cria uma barreira para evitar que tarefas de redução comecem antes de todas as tarefas de mapeamento acabarem. Os *workers* então realizam procedimentos remotos para ler estes dados e quando todos os dados são lidos, eles são ordenados pelas suas chaves de forma que todas as ocorrências de uma mesma chave fiquem juntas. O trabalho de redução então é executado e um novo par de chave e valor é gerado, e seus resultados salvos em um sistema de arquivos distribuído. Após todo o fluxo de execução do modelo de *MapReduce* é retornado um arquivo para cada *worker* que realizou uma tarefa de redução (DEAN et al., 2004).

Como o modelo de *MapReduce* foi criado para gerir o processamento de grandes quantidades de dados distribuídos em milhares de máquinas, são empregadas algumas técnicas de tolerância a falhas. O *master* envia periodicamente aos *workers* mensagens de ping, se ele não receber uma resposta em um tempo determinado, este trabalhador é marcado e todas as tarefas designadas a ele são marcadas para uma nova distribuição. Este mecanismo funciona tanto para tarefas de mapeamento como de redução. Já o *master* possui um mecanismo de salvamento periódico do estado do fluxo de execução, se o processo do *master* morrer ele é reiniciado do último salvamento.

2.1.2 Apache Flink

Apache Flink é um *framework* de *BigData* para processamentos *stream* e *batch* de código aberto. Desenvolvido pela Apache *Software Fundation*, o seu modelo de programação de fluxo de dados provê processamento tanto para fluxos de dados finitos como para fluxos potencialmente infinitos. Sua arquitetura de software pode ser vista na figura 2.2. No modo de processamento de fluxo, em geral os dados de entrada tendem a ser gerados de uma forma contínua e em tempo real, em geral avaliados por meio de janelas do fluxo de dados. Já o modo de processamento em lotes pode ser pensado como um caso limitado do processamento de fluxo em que a janela de execução dos dados abrange o tamanho total do arquivo de entrada e o fluxo de execução não depende da geração de dados em tempo de execução.

Figura 2.2: Representação *Bottom-up* Apache Flink



Fonte: (FLINK, 2018).

O *framework* conta com APIs escritas em linguagens como Java, Scala e Phython que auxiliam sua aplicação para os mais diversos problemas. As duas principais APIs do *framework* são as API de *DataStream* que implementa transformações em fluxos de dados e a

API de *DataSet* que implementa transformações em lotes de dados. Outras bibliotecas de funções para propósitos específicos também estão presentes no *framework* como a de expressões em SQL para tabelas relacionais (*Table*), a de Aprendizado de Máquina (*FlinkML*) e a de processamento de grafos (*Gelly*). O núcleo de execução do *framework* é um mecanismo de fluxo de dados de streaming distribuído, o que significa que os dados são processados um evento por vez e não como uma série de lotes, isso permite que o *framework* utilize muitos recursos de resiliência e desempenho. Flink pode ser executado em modo local rodando em uma máquina virtual Java ou ainda em *clusters* distribuídos e *clouds* (CARBONE, 2015).

2.1.2.1 Caracteristicas

As características descritas abaixo são as que permitem que o *framework* Flink retorne resultados precisos para o processamento de grandes fluxos de dados. Flink foi pensado para rodar em clusters e realizar computação em qualquer escala.

Flink garante que os resultados serão gerados com o processamento de todas as partes do arquivo de entrada exatamente uma vez. Isso é feito por mecanismos que salvam pontos de recuperação da aplicação para casos de falhas na execução. Quando necessário estes pontos são recarregados e utilizados para continuar a computação em algum caso de erro. Flink também suporta janelas de utilização dos dados, ordenadas por eventos de tempo, ou seja, ele garante que eventos que chegam fora de ordem, mas que precisem ser processados com outros eventos na mesma janela de tempo, sejam agrupados conjuntamente. Isso é de extrema importância em problemas em que a ordem que os eventos ocorrem influência nos resultados obtidos. A flexibilidade da customização das janelas de utilização dos dados não se restringe apenas ao tempo dos eventos, mas também a contagens, sessões e janelas influenciadas pelo fluxo dos dados, o que permite que aplicações fiquem mais próximas dos problemas reais que queremos resolver.

Para o *framework* executar aplicações no modo de lotes de arquivos, ele trata isso como uma execução especial do modo de fluxo de dados. O *Dataset* do arquivo de lote é tratado internamente como um *DataSet* de *stream*, e os conceitos aplicados à execução em stream também são aplicados a este caso específico. O esquema de tolerância a falhas presente no Flink é considerado de baixo custo para a execução, permitindo que o framework

consiga manter uma alta vazão de dados de saída e garantindo a consistência dos resultados gerados. Ele consegue se recuperar de falhas sem nenhuma perda de dados.

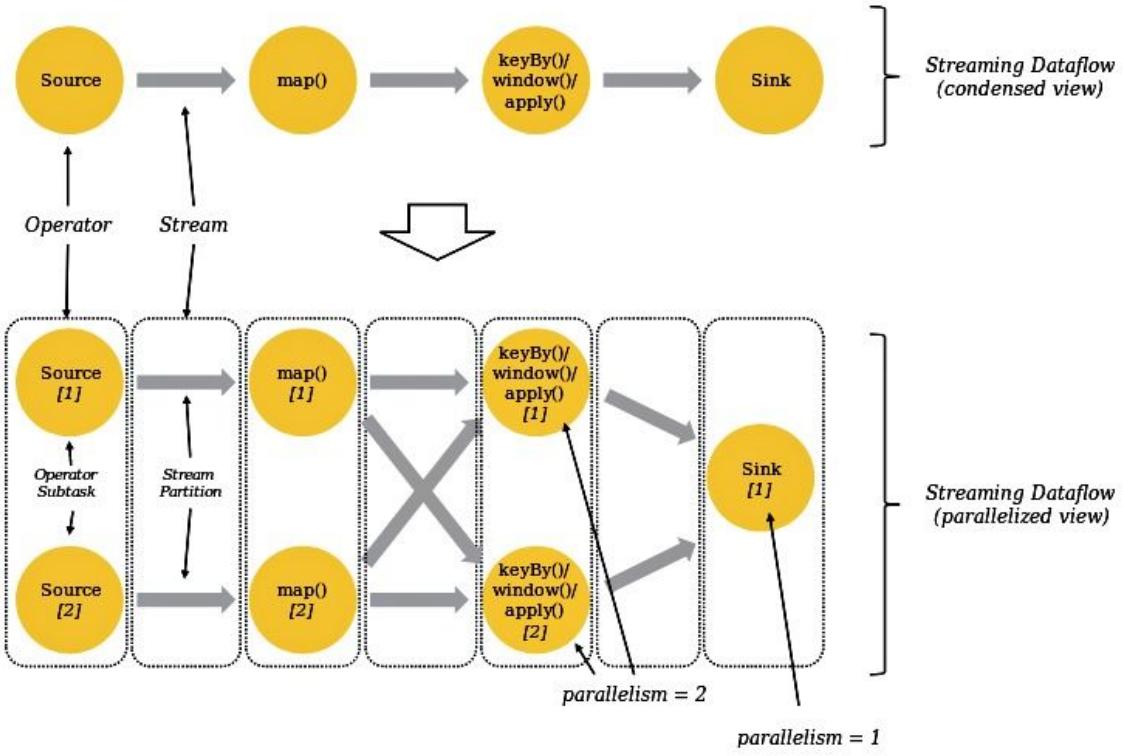
No *framework* Flink a figura do master é chamada pelo nome de *JobManager*. Ela é encarregada de toda a parte de gerência do processamento dos dados, salvando de pontos de recuperação e encaminhando trabalho para os *workers*, chamados de *TaskManagers*. Aos *TaskManagers* ficam as tarefas de executar as funções de mapeamento e redução, além de informar ao *JobManager* seu estado de execução periodicamente.

2.1.2.2 Fluxos de dados e Programas

Basicamente programas em Flink são constituídos por blocos de transformações e streams, onde streams se referem a um fluxo de dados (que potencialmente nunca termina) e transformações as operações que processam estes dados. Cada um destes fluxos de dados começa com uma ou mais fontes de dados (*Source*) e acaba em uma ou mais saídas destes dados (*Sink*). Na figura 2.3 é possível notar que o fluxo lembra muito o de um Grafo Acíclico Direcionado (*Directed Acyclic Graph* - DAG), mas formas especiais de ciclos são permitidos no *framework* via construtores de iterações. A forma mais comum das transformações é a em que um dado de entrada gera um dado de saída, entretanto existem casos em que as transformações podem gerar para cada dado de entrada, zero ou mais dados de saída.

Programas escritos utilizando Flink são inerentemente paralelos e distribuídos. Durante a execução da aplicação, cada fluxo de execução possui uma ou mais partições de *stream* e cada operador possui uma ou mais subtarefas, onde uma tarefa é uma operação de map, reduce, source, sink, etc e uma subtarefa é uma divisão desta tarefa. Cada operador de subtarefa opera independentemente dos outros, sendo executados em threads diferentes e possivelmente em máquinas diferentes. O número de operadores de subtarefas é equivalente ao paralelismo empregado na tarefa, mas diferentes estágios do fluxo de execução podem empregar diferentes níveis de paralelismo, como em tarefas de sink onde podemos querer que todos os dados processados sejam salvos juntos. O fluxo de por onde os dados passa não precisa ficar contido em um único operador, de um estágio do fluxo de execução para outro, ou seja, de uma transformação para outra, os dados intermediários podem ser redistribuídos entre os operadores para agrupar juntos dados que se relacionam e que na próxima transformação devem ser processados juntos.

Figura 2.3: Fluxo de Execução simples e paralelo dos dados no Apache Flink



Fonte: (FLINK, 2018).

2.1.2.3 Principais Funções

O *framework* possui uma variedade muito grande de funções implementadas para abranger um grande número de problemas que os usuários dele possam ter. A seguir iremos descrever apenas as principais funcionalidades e as utilizadas no decorrer deste trabalho. A API escolhida foi a *DataSets*, para processamento de dados em lotes (*batch*).

Nas funcionalidades de entrada de dados foram utilizadas a função *readTextFile*, onde se lê do disco um arquivo de texto e o transforma em strings, e a função de *readCsvFile* onde os arquivos de entrada possuem uma estrutura mais definida e os dados contidos no arquivo, são separados por um delimitador que pode ser passado para a função. Em ambos os casos os dados são transformados em *DataSets*, que são estruturas criadas pelo *framework* para manipular os dados intermediários, de entrada e de saída de uma aplicação.

Para as tarefas de transformações várias funções foram utilizadas. Nos trabalhos de mapeamento foram amplamente utilizadas as funções de *map*, que retorna para cada dado de entrada um dado de saída, e a função de *flatMap*, que possui uma implementação mais livre e

permite que para cada de entrada possam ser gerados zero ou mais dados de saída. Em ambas as funções a lógica é definida pelo programador. Para a filtragem de dados foi utilizada a função de *filter* que filtra os dados de acordo com uma função booleana definida pelo programador. Para agregação dos dados foi utilizada as funções de *groupBy*, que une dados por um ou mais campos passados para a função, e as funções de *where* e *equalTo* que juntas com a função *join* unem dois *DataSets* diferentes em um único de acordo com uma chave em comum definida. No processamento das reduções a mais utilizada foi a função de *reduce*, que tem sua lógica definida pelo programador e que reduz um grupo de elementos para um único repetidamente combinando dois em um. Também foi utilizada a função de redução *sum*, previamente programada, e que pode ser utilizada em uma variedade de tarefas. Ela recebe um *DataSet* previamente agregado e como parâmetro da função qual elemento deste *DataSet* queremos somar.

Outra funcionalidade utilizada foi a *withBroadcastSet* que envia um *DataSet* previamente carregado a todas as instâncias que irão processar alguma transformação e necessitem deste *DataSet*. Esta funcionalidade tem um grande impacto no desempenho do processamento pois se utilizada com arquivos muito grandes pode gerar um tráfego de dados alto entre os nós que forem designados para o trabalho.

2.2 PLN

O Processamento de Linguagem Natural (PLN) é uma área da computação que é utilizada em conjunto com várias outras áreas, com destaque para a inteligência artificial, linguística e ciência da informação. Ela estuda meios computacionais capazes de interpretar e gerar informações a partir de dados em linguagem natural. Para isso ser possível várias características da linguagem devem ser levadas em conta como sons, palavras, sentenças, estruturas gramaticais, entre outras (VIEIRA, 2001). Este campo da computação sempre recebeu muita atenção pois ele possui o grande desafio de conseguir fazer com que máquinas entendam a fala humana, e a partir disso possam seguir instruções.

Podemos dividir o Processamento de Linguagem Natural em níveis de análise como a análise morfológica, análise sintática, análise semântica, análise de discurso e a análise pragmática. Na análise morfológica o objetivo é identificar palavras (*tokens*) ou expressões

isoladas em uma sentença, com o auxílio de delimitadores (espaços em branco e pontuação). As palavras são identificadas e classificadas de acordo com o problema a ser resolvido. A análise sintática consiste em criar árvores de derivação para as sentenças e palavras obtidas no analisador morfológico e mostrar como elas se relacionam umas com as outras. Esta análise tem como um dos grandes desafios contornar as ambiguidades presentes em linguagens naturais, transformando elas em algo que possa ser interpretado corretamente por um computador. Na análise semântica verificamos o sentido das palavras que foram agrupadas pelo analisador sintático. Compreender a relação entre as palavras é um importante passo na análise das linguagens naturais e mais uma vez a ambiguidade pode se tornar um problema. Nas sentenças "tomar uma cerveja" e "tomar banho" a mesma palavra tem significados diferentes que devem ser resolvidos nesta fase. Na análise de discurso verificamos se frases anteriores ou posteriores podem sinalizar o significado de uma outra frase ou palavra. Aqui analisamos o contexto onde as informações estão inseridas. Por fim temos a análise pragmática onde verificamos as informações como um todo e não apenas partes separadas, determinando o que está querendo ser transmitido nos dados analisados (LIDDY, 2001).

2.2.1 Similaridade entre Palavras

O significado de uma palavra desconhecida pode ser inferido pelo seu contexto (LIN, 1998). Para construir modelos de medição da similaridade entre palavras, duas principais abordagem são em geral aplicadas. A abordagem mais tradicional é baseada em contagens de *tokens* onde a informação é extraída diretamente do texto utilizado (LIN, 1998) ou utilizando medidas associativas como a Informação Mútua Pontual (CHURCH; HANKS, 1990). Mais recentemente a abordagem preditiva vem ganhando espaço, representando palavras como vetores criados através de redes neurais (MIKOLOV et al., 2013a). Em ambos os casos palavras são representadas como vetores que carregam informações sobre o contexto escolhido.

2.2.2 Representação Distribucional

Representações distribucionais de palavras são baseadas em matrizes de coocorrência. A matriz possui um tamanho $W \times C$, onde W é o tamanho do vocabulário do texto analisado e

C o tamanho do contexto utilizado (TURIAN et al., 2010). As linhas da matriz carregam informações sobre as palavras enquanto as colunas sobre o contexto, estas matrizes têm como características serem grandes e esparsas. Existem muitos fatores que influem no resultado final da geração e devem ser escolhidos para criar as representações (TURNEY, 2010), como o tipo e o tamanho do contexto a ser utilizado, o tamanho do vocabulário, etc. Em geral a escolha destes parâmetros depende do problema a ser abordado e suas peculiaridades, algo que funciona bem para uma solução pode não dar resultados satisfatórios para outra.

Guardar e utilizar estas matrizes pode se tornar uma tarefa desafiadora do ponto de vista computacional. Se utilizarmos textos muito grandes e contextos muito abrangentes, o tamanho desta matriz tende crescer até um ponto onde não é mais viável sua manipulação. Um método utilizado para gerar matrizes menores que contenha a informações da similaridade de palavras e contextos é o mapeamento da matriz *F* de tamanho *W x C* para uma matriz *H* de tamanho *W x d* onde $d << C$ (TURIAN et al., 2010).. Para isso devemos escolher uma função g de forma que $H = g(F)$.

2.2.3 Informação Mútua Pontual

Matrizes de coocorrência são geradas usualmente pela soma das vezes em que uma palavra apareceu perto de uma outra. Apesar de parecer uma forma satisfatória de geração deste tipo de matriz o método da frequência não é a melhor forma de medir a associação entre palavras (JURAFSKY, 2008). Um dos grandes problemas é que a contagem simples da frequência não é muito discriminativa. Palavras que ocorrem muitas vezes em um texto como mas, eles, nós, etc, aparecem geralmente próximas de todos os tipos de palavras, fazendo com que a contagem de frequência perca um pouco da sua capacidade de discriminar precisamente a relação entre as ocorrências.

Um método muito divulgado e aceito para medir estas matrizes é a Informação Mútua Pontual (Pointwise Mutual Information - PMI). Proposto por Church e Hanks (1989) e (Church e Hanks, 1990), ele se baseia na noção informação mútua. Na prática ele é a medida de quão frequente dois eventos A e B ocorrem comparado com o que esperaríamos se estes eventos fossem independentes (Fano, 1961). Aplicando este pensamento para vetores de

coocorrência podemos definir a informação mútua pontual da associação entre uma palavra ‘*w’* e uma palavra de contexto ‘*c’* como:

$$PMI(w,c) = log_2 \frac{P(w,c)}{P(w)P(c)}$$

No numerador temos a probabilidade das duas palavras ocorrerem juntas, já no denominador temos a multiplicação da probabilidade da palavra *w* ocorrer pela probabilidade da palavra *c* ocorrer no texto de forma independente.

2.2.3.1 Informação Mútua Pontual Positiva

O valor da equação PMI varia de valores positivos para valores negativos tendendo ao infinito. Como valores negativos nesta estimativa não fazem sentido, pois uma palavra não pode ter um número negativo de ocorrências em relação a uma outra, é muito comum na área ser utilizado uma variação do PMI que substitui todos os valores negativos do PMI por zero, sendo chamado de *Positive Pointwise Mutual Information* (PPMI) (DAGAN et al., 1993). Ele pode ser definido como:

$$PPMI(w,c) = max(log_2 \frac{P(w,c)}{P(w)P(c)}, 0)$$

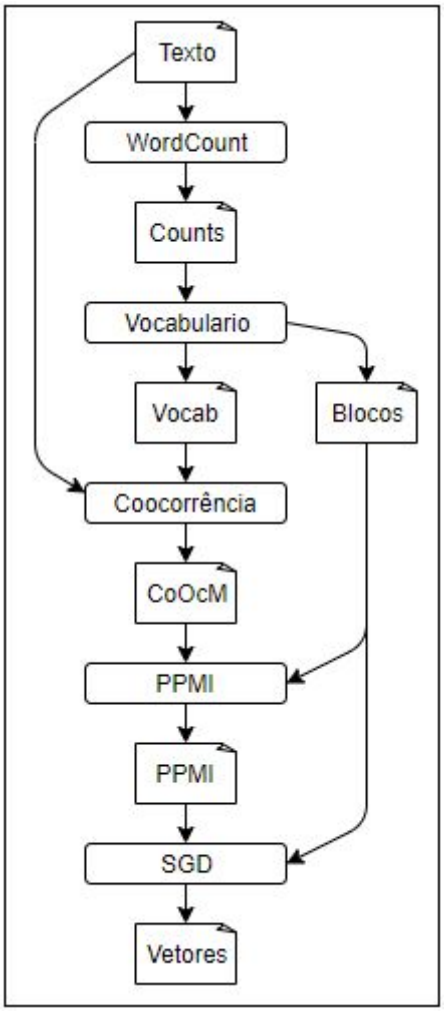
2.3 Considerações Finais

Neste capítulo foram descritos os conceitos que serviram de base para o trabalho desenvolvido. Na seção de *BigData* foram tratados os conceitos básicos da área, do modelo de *MapReduce* e do *Framework* Flink, utilizados no trabalho com o objetivo de conseguirmos paralelizar o problema da fatoração de grandes matrizes. Na seção seguinte os conceitos de Processamento de Linguagem Natural, de representações distribuídas e da geração de matrizes de coocorrência são apresentados, foco principal do trabalho e que serão utilizados no próximo capítulo, onde será definido o modelo proposto.

3 MODELO

Neste capítulo será descrito o modelo da solução para o problema da fatoração de uma matriz de representação distribuída de palavras. O processo foi dividido em duas etapas: o pré-processamento e a etapa de fatoração da matriz PPMI pelo gradiente descendente estocástico (SGD). No pré-processamento o texto de entrada é analisado e transformado. Este processo foi dividido em quatro tarefas que serão detalhadas na seção 3.1. A etapa de SGD é responsável pela geração do resultado final e será descrita na seção 3.2, sendo esta a etapa mais onerosa em termos de tempo de processamento e complexidade. Na seção 3.3 são apresentadas as considerações finais do capítulo. A figura 3.1 mostra o fluxo de execução desde o arquivo de entrada até a saída final do processamento.

Figura 3.1: Fluxo de execução do modelo proposto.



3.1 Pré-Processamento

No pré-processamento o texto de entrada é transformado em dados estruturados que servirão de entrada para o cálculo da fatoração da matriz PPMI.

3.1.1 WordCount

Tarefa inicial da solução do problema, esta etapa recebe como entrada um texto em linguagem natural. A tarefa divide o texto em frases que são processadas paralelamente. Estas frases são divididas, por caracteres de espaços em branco e pela pontuação, para então encontrar as palavras contidas no texto. Tuplas contendo as palavras encontradas são emitidas, agrupadas e somadas retornando um arquivo com as tuplas de palavras e o número de vezes que elas ocorrem no texto de entrada. Um exemplo deste processamento utilizando a frase "a casa da esquina é maior que a minha casa." geraria as seguintes tuplas: (a,2), (casa,2), (esquina,1), (é,1), (maior,1), (que,1) e (minha,1).

3.1.2 Vocabulário

Nesta tarefa construímos o vocabulário a ser utilizado nas tarefas seguintes. O arquivo gerado na etapa do WordCount é utilizado como entrada desta etapa. Este arquivo é filtrado, excluindo palavras que aparecem poucas vezes no texto, para este trabalho foi escolhido filtrar palavras que possuam menos de 100 ocorrências no texto (SALLE et al., 2016a). Como saída são gerados dois arquivos, o primeiro com o vocabulário filtrado contendo as tuplas formadas por palavra e o número de ocorrências dela no texto. O segundo arquivo gerado contém as palavras do vocabulário, já filtradas, divididas em n blocos de processamento, onde n é o paralelismo máximo a ser empregado na última etapa do modelo. Este segundo arquivo servirá de entrada em mais uma etapa do pré-processamento e também na etapa de SGD, onde auxiliará na distribuição dos vetores das palavras que serão fatorados pelos workers. Mais detalhes da sua utilização serão descritos nas próximas seções deste capítulo.

3.1.3 Coocorrência

Nesta tarefa são contadas todas as coocorrências presentes no texto, para as palavras do vocabulário gerado no passo anterior. Esta etapa recebe como entrada além do arquivo de vocabulário filtrado, o arquivo de texto original que foi utilizado na etapa de *WordCount*. O processamento é realizado analisando o texto frase por frase, removendo todas as palavras que não estão presentes no vocabulário filtrado. As frases então são novamente avaliadas, procurando as coocorrências existentes para cada palavra separadamente, considerando uma janela pré-definida de palavras a direita e à esquerda da palavra avaliada. Neste trabalho foi utilizado uma janela de duas palavras a direita e duas palavras à esquerda da palavra analisada (SALLE et al., 2016a). Para cada coocorrência encontrada, uma tupla é gerada, contendo a palavra e a palavra de contexto. Como passo final, estas tuplas são agrupadas em coocorrência iguais e somadas, retornando um arquivo contendo tuplas com as coocorrência existentes e o número de vezes que elas se repetem no texto.

Um segundo processamento é realizado nesta etapa, a geração de coocorrências negativas. Para cada palavra presente no vocabulário, são geradas coocorrência aleatoriamente com outras palavras presentes no vocabulário, com a intenção de adicionar relações que não existem no texto, com um valor nulo. O número de coocorrências negativas geradas, para cada palavra, é definido pela seguinte fórmula arredondada para cima:

$$N = w * 5 * (1 - \sqrt{t/w/c})$$

onde w é o número de vezes que uma palavra aparece no texto, c o número total de palavras no texto e t um *threshold*. A fatoração da matriz PPMI em tarefas de similaridade de palavras pode ser melhorada utilizando este método de adição de coocorrências que não acontecem no texto com um valor igual a zero (MIKOLOV et al., 2013b). Para este trabalho iremos utilizar um *threshold* de t = 10^-5 (MIKOLOV et al., 2013b).

Ao final deste segundo processamento os dois arquivos são unidos em um único, contendo tuplas formadas pelas coocorrências encontradas. Estas tuplas contêm a palavra, a palavra de coocorrência, o número de coocorrências positivas destas duas palavras e o número total de coocorrências, somando as ocorrências negativas e as positivas.

3.1.4 PPMI

Tarefa final do pré-processamento, nesta etapa é calculada a informação mútua pontual positiva de cada coocorrência encontrada na etapa anterior. O cálculo segue a fórmula apresentada no capítulo 2 deste trabalho. Para as probabilidades foram utilizadas as contagens de palavras e coocorrências. Como entrada temos o arquivo com as tuplas contendo as coocorrências e as contagens positiva e negativas; para o cálculo do PPMI utilizamos apenas o valor das coocorrências positivas encontradas no passo anterior.

Nesta etapa dividimos todas as tuplas de coocorrência em setores, para que na etapa seguinte todas as tuplas de um mesmo setor possam ser agrupadas e processadas sequencialmente. O arquivo com as palavras do vocabulário divididas em blocos, gerado na etapa de Vocabulário, é utilizado para mapear as tuplas para os setores, de acordo com a posição das cada palavras e contexto nos blocos. Como resultado geramos um arquivo que contém tuplas contendo além das coocorrências, suas contagens positivas e negativas, o valor PPMI calculado e o setor ao qual a tupla pertence.

3.2 Gradiente Descendente Estocástico

O Gradiente Descendente Estocástico (*Stochastic Gradient Descent* - SGD) é um método de otimização muito utilizado em aprendizado de máquina. Seu objetivo é achar um valor $\theta^* \in \Re^k$ ($k \geq 1$) que minimiza uma dada perda $L(\theta)$. Começando com uma valor inicial θ_0, o SGD refina o valor do parâmetro iterando a diferença estocástica da equação

$$\theta_{n+1} = \theta_n - \varepsilon_n L'(\theta_n),$$

onde n é o número do passo da iteração e $\{\varepsilon_n\}$ é a sequência decrescente do tamanho do passo, assumindo que ε_n é não negativo e finito (GEMULLA et al., 2011). A teoria da aproximação estocástica pode ser utilizada para mostrar que, em certas condições regulares (KUSHNER, YIN, 2003), o ruído do gradiente estima a "média" e o SGD converge para pontos estacionários onde $L'(\theta)=0$. Estes pontos estacionários podem ser pontos de mínimos ou máximos locais. Para conseguir encontrar os pontos de mínimo ou máximo globais uma variedade de técnicas pode ser utilizada, como por exemplo rodar o SGD várias vezes ou começar por pontos escolhidos randomicamente.

Neste trabalho iremos utilizar o SGD a fim de aproximar uma matriz *M(V,V)* utilizando duas matrizes menores *W(V,d)* e *C(V,d)* onde *V* é o vocabulário utilizado, *d* é a dimensão dos vetores desejados e d << V. A função de aproximação utilizada é a seguinte:

$$g = W_a C_b^T - PPMI_{ab}$$

Na equação acima calculamos o erro da diferença entre o produto vetorial, do vetor da palavra de vocabulário (W_a) multiplicado pelo vetor transposto da palavra de contexto (C_b), e o valor PPMI desta coocorrência. Este erro então é multiplicado por uma taxa de aprendizado e subtraído de todas as posições dos vetores. Este processo é repetido pelo número de coocorrências positivas e negativas encontrados. O método depende dos passos anteriores para conseguir calcular os próximos a fim de aproximar os vetores, gerados randomicamente no início do processo, da matrix PPMI, calculada no pré-processamento.

3.2.1 Gradiente Descendente Estocástico Distribuído

Para conseguirmos paralelizar e distribuir este processo, foi necessário quebrar os vetores de palavras em blocos e a matriz PPMI em setores. Como precisamos do valor anterior calculado para cada vetor, dividimos o processamento em *n* passos, onde *n* é o paralelismo empregado e também o número de blocos de vetores que teremos. Já a matriz terá sempre n^2 setores que serão mapeados de acordo com a posição da palavra e do contexto nos blocos de vetores; esses setores podem ser divididos em grupos que são processados paralelamente. Em uma iteração do método cada tupla da matriz PPMI, de um setor específico, é unida com as outras tuplas do mesmo setor. Isso também é feito para os vetores de palavras e de contextos, que estão presentes nas tuplas da matriz, e serão utilizados na aproximação dos valores PPMI. Estas tuplas de setores e dos vetores que foram unidas, são então enviadas ao *worker* que irá realizar a etapa de SGD sequencialmente neste determinado setor. O cálculo do SGD é realizado em cada setor de um grupo paralelamente e este processo se repete n vezes até que todos os setores sejam processados.

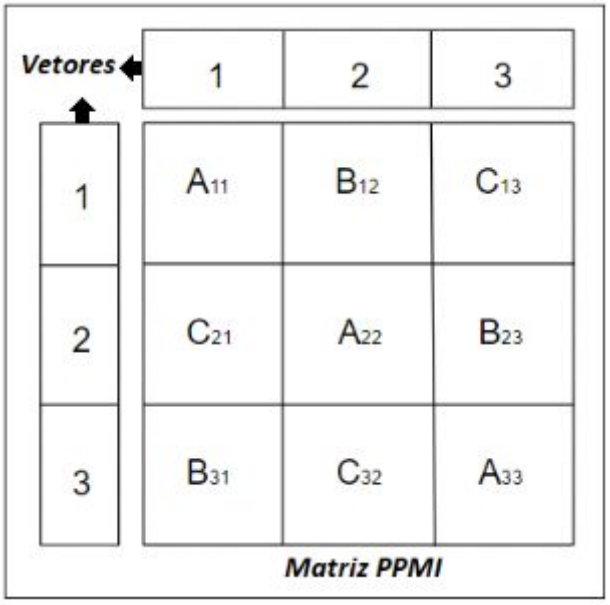
A cada etapa do método cada fluxo de execução recebe um bloco de tuplas de vetores das palavras, um bloco de tuplas de vetores dos contextos e um setor da matriz com as tuplas geradas no último passo do pré-processamento. O *worker* então aplica o SGD neste setor utilizando os blocos de vetores que ele possui. Ao final deste passo são retornados os blocos

de vetores das palavras e dos contextos, com os valores calculados até aquele ponto, e o erro de cada iteração, que é calculado pelo seguinte fórmula:

$$e = e + g^2 / 2$$

Uma iteração do SGD só é completada quando todos os setores tiverem sido processados uma vez por algum *worker*. O método deve ser repetido por várias vezes, buscando sempre a cada iteração diminuir o erro desta aproximação. Quando o erro para de diminuir ou sua variação é muito pequena a execução do modelo acaba e podemos avaliar os resultados.

Figura 3.2: Divisão de Vetores de Palavras e Matriz PPMI com um n=3.

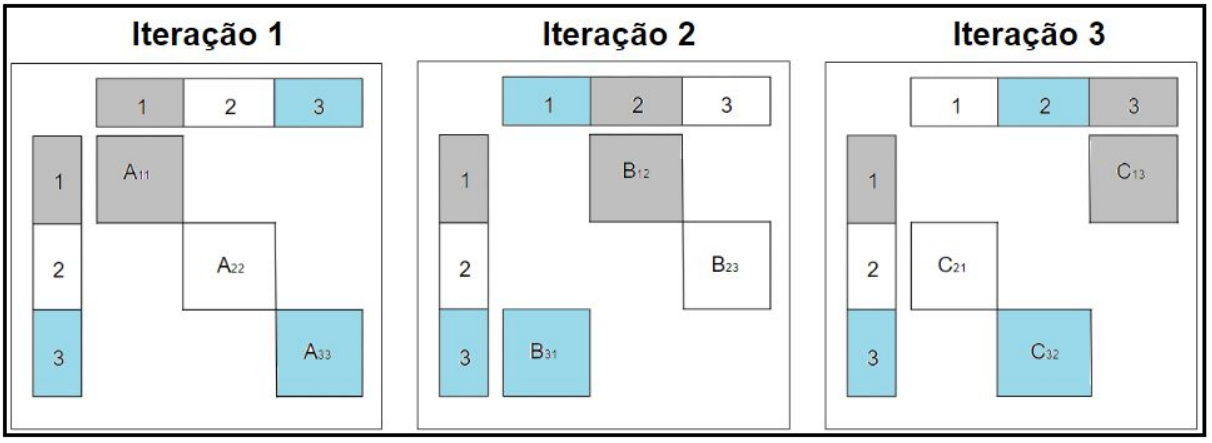


Um exemplo desta divisão com n=3 foi desenhado na figura 3.2. Neste exemplo podemos ver que a matriz foi dividida em 9 partes nomeadas de acordo com os grupos de setores que serão processados paralelamente e com os números dos blocos de vetores que serão utilizados nestes grupos. Na figura 3.3 foi desenhado as 3 iterações necessárias para executar a etapa do gradiente descendente estocástico, de forma paralela em todos os pontos da matriz PPMI, do exemplo da figura 3.2. Na primeira iteração os setores *A* são distribuídos para os *workers* responsáveis pela sua execução, junto aos blocos de vetores de palavras e contextos. Na segunda iteração os setores calculados na etapa anterior são novamente distribuidos paras os workers, mas agora com os setores nomeados *B*. Na 3 iteração do modelo os setores nomeados *C* são distribuídos aos *workers* junto aos vetores calculados na

iteração 2. Quando as 3 iteração são concluídas o modelo foi executado em toda a matriz PPMI.

.

Figura 3.3: Todas as iterações de uma divisão de blocos com n=3.



3.3 Considerações Finais

Neste capítulo foi descrito o modelo proposto para paralelizar a fatoração de grandes matrizes utilizando o gradiente descendente estocástico (SGD) a partir de uma matriz de informação mútua gerada com informações retiradas de um texto de entrada. O modelo como explicado acima pode ser implementado em qualquer *Framework* de *MapReduce*. No próximo capítulo será detalhado como foi implementado o modelo no *Framework* Apache Flink.

4 PROTÓTIPO

Este capítulo descreve como o modelo apresentado e descrito no capítulo 3 foi implementado. Na seção 4.1 são relatados a linguagem e o *framework* utilizados. A seção 4.2 descreve características da implementação e decisões tomadas. Por fim na seção 4.3 são feitas considerações finais sobre a implementação e são discutidos os problemas enfrentados durante o trabalho.

4.1 Linguagens e Frameworks

A solução foi inteiramente escrita utilizando o *Framework* Apache Flink e a linguagem de programação Java. A escolha pela linguagem se deu pelo suporte nativo que o *framework* fornece ao Java. Para auxiliar a implementação do trabalho foi utilizado o software de gerenciamento de projetos Maven[5], que foi inicialmente desenvolvido para simplificar a construção de processos no projeto Jakarta Turbine. Neste trabalho o Maven auxiliou na compilação e na organização do projeto e de suas dependências.

4.2 Características da Implementação

A implementação da solução foi feita na sua grande parte no modelo de MapReduce. Todas as etapas do pré-processamento e a etapa do SGD conseguiram ser adaptadas ao modelo e consequentemente paralelizadas e distribuídas. A única parte que não foi possível adaptar ao modelo foi a união dos arquivos contendo as coocorrências positivas e negativas. Por ser necessário somar dois campos diferentes na mesma operação, está parte foi implementada sequencialmente na linguagem Java.

As etapas do pré-processamento, implementadas no modelo *MapReduce*, podem ter qualquer nível de paralelismo, sendo limitadas apenas pelas características do Framework e dos arquivos de entrada. A etapa de SGD tem sua paralelização vinculada ao arquivo gerado

[5] Apache Maven - https://maven.apache.org/. Acessado 2 Jun. 2018.

na etapa de Vocabulário, que divide as palavras em blocos para serem distribuídas entre os *workers*, responsáveis pela computação. Se utilizarmos uma paralelização maior que o número de blocos gerados, alguns workers ficaram sem trabalho já que a matriz PPMI é dividida em setores de acordo com o número de blocos, neste caso o nível de paralelismo será igual ao número de blocos. Já se a paralelização for menor que o número de blocos a solução executará normalmente e alguns *workers* ficarão responsáveis por processar um número maior de setores da matriz.

O texto de entrada precisa ser processado antes da sua utilização para normalizar sua estrutura e resolver algumas ambiguidades. Palavras iguais mas que possuam algum caractere em maiúscula são interpretadas como palavras diferentes. O mesmo pode ocorrer com palavras escritas com ou sem algum acento.

O erro extraído em cada etapa do processamento do SGD não pode ser coletado diretamente da computação pois o modelo de *MapReduce* implementado no *Framework* Flink só consegue retornar, para cada sequência de transformações, um único arquivo de saída. Para contornar este problema foi criada, no fim do processamento de um setor da matriz, uma tupla especial que carrega este erro para fora do processamento. Esta tupla é formada por caracteres especiais que não fazem parte de nenhuma palavra em linguagem natural. O erro acumulado é então colocado na primeira posição do vetor desta nova palavra especial e o resto do vetor é preenchido com valores nulos. Este erro então é analisado, fora do processamento do SGD, para avaliar o andamento do resultado e na próxima iteração do método esta tupla especial é descartada.

Para tentar escapar de mínimos locais na fatoração da matriz PPMI os vetores utilizados na primeira iteração da etapa de SGD são gerados aleatoriamente (GEMULLA et al., 2011). Nas iterações seguintes, os vetores gerados na etapa anterior alimentam a próxima, com o objetivo de diminuir o erro coletado e gerar vetores que se aproximem ao máximo da matriz PPMI calculada.

4.3 Considerações Finais

Neste capítulo foram descritos as características principais da implementação do modelo proposto no capítulo 3. Apesar de o modelo ter sido quase que inteiramente

implementado no modelo de *MapReduce*, dificuldades de utilização de memória e de adaptar certas partes do problema ao *Framework* foram enfrentadas. A etapa mais complicada, e foco principal do trabalho, foi a da iteração do método de SGD, por ser a etapa que deve manipular a grande matriz, o uso de memória RAM das máquinas utilizadas é muito alto, o que fez com que a implementação desta etapa fosse dividida, salvando o resultado intermediário, de uma iteração em um grupo de blocos, periodicamente em disco. Esta decisão fez com que o desempenho da etapa de SGD caísse, mas possibilitou a manipulação de textos maiores. No próximo capítulo a metodologia e os resultados dos testes realizados serão apresentados.

5 RESULTADOS

Neste capítulo serão apresentados os resultados obtidos nos experimentos realizados. Na seção 5.1 apresentamos a metodologia utilizada para avaliar a aplicação e . Na seção 5.2 descrevemos as máquinas utilizadas detalhando os seus respectivos hardwares e softwares. Finalmente na seção 5.3 os tempos coletados de cada uma das etapas são detalhados.

5.1 Metodologia de Avaliação

Os experimentos foram realizados para avaliar o desempenho da aplicação com base no paralelismo empregado, coletando o tempo necessário para finalizar a execução de cada um dos experimentos. Para um maior detalhe dos resultados de todo o modelo proposto, todas as etapas do pré-processamento e a etapa de SGD foram avaliadas individualmente.

Com os tempos obtidos foi calculado o *speedup* e a eficiência de cada experimento. O *speedup* ajuda a avaliar o aumento de desempenho da aplicação quando aumentamos o paralelismo. Ele é definido pelo tempo para se processar a aplicação com apenas um fluxo de execução dividido pelo tempo obtido com algum nível de paralelismo. Já a eficiência avalia a utilização dos processadores, e podemos calcular ela dividindo o *speedup* encontrado pelo número de processadores utilizados. Como dados de entrada utilizamos o Europarl[6], que reúne as atas dos procedimentos do parlamento europeu.

5.2 Ambiente dos Experimentos

O ambiente utilizado nos experimentos era formado por uma máquina virtual instanciada na plataforma da Microsoft Azure. A máquina possuía processador Intel Xeon E5-2673 de 2.4 GHz com 16 GB de memória RAM e 8 núcleos de processamento. O sistema operacional da máquina era Ubuntu 16.04, o framework Apache Flink utilizado estava na versão 1.4.2 e a Máquina Virtual Java na versão 1.8.

[6] Eroparl, http://www.statmt.org/europarl/. Acessado 2 Jun. 2018.

5.3 Tempos Coletados

Para os experimentos foram utilizados arquivos de 1 GB, 3 GB e 6 GB de textos do *Europarl*, que foram preparados para serem utilizados pela aplicação. Todos os caracteres que não eram letras foram retirados, ficando somente as palavras e os caracteres de nova linha. Os textos analisados possuem um total de 170, 480 e 950 milhões de palavras ao total respectivamente.

5.3.1 Pré-Processamento

Nesta seção serão apresentados os tempos coletados nos experimentos da etapa de pré-processamento e as análises realizadas.

5.3.1.1 WordCount

Na figura 5.1 temos os tempos obtidos ao executar a etapa de WordCount variando o paralelismo utilizado nos arquivos de entrada. Já na figura 5.2 temos o *speedup* destas execuções e na figura 5.3 a eficiência. Analisando os resultados observados nas imagens é possível notar que, utilizando arquivos maiores, o *speedup* e a eficiência do uso dos processadores mostra uma tendência a decrescer mais lentamente. Isso acontece pois o processo de contar as palavras de um texto é um problema que se molda muito bem ao modelo de MapReduce. Na função de mapeamento é possível dividir facilmente os dados de entrada pelas linhas presentes no texto, processando cada uma delas individualmente, e na etapa de redução, palavras iguais coletadas no passo anterior são enviadas para os mesmos *workers* para serem agregadas e somadas.

Figura 5.1 Tempo de execução WordCount

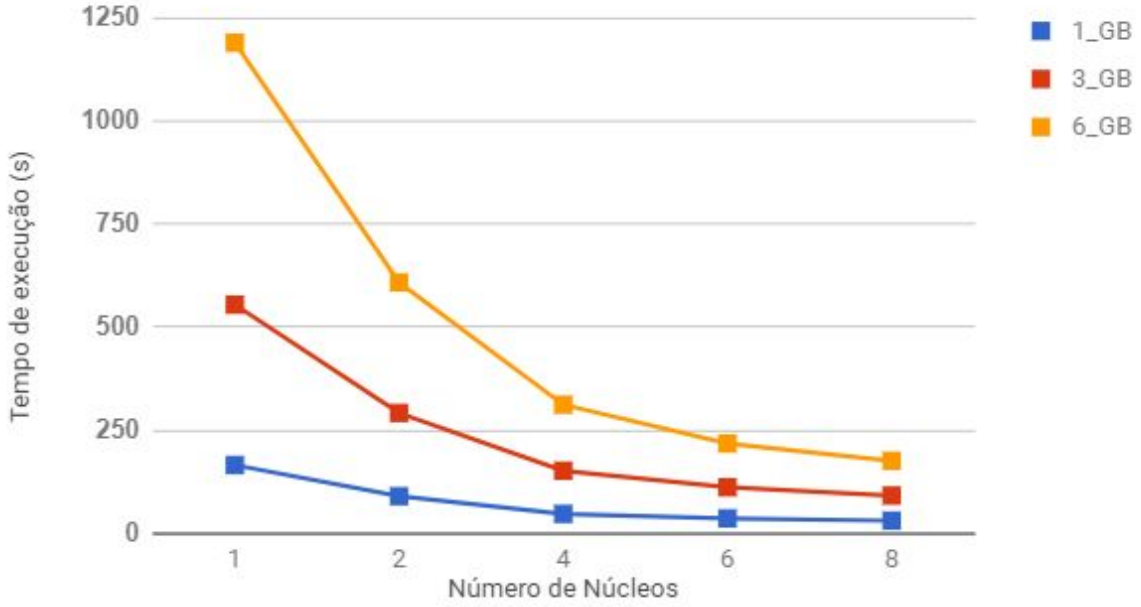


Figura 5.2 *Speedup* WordCount

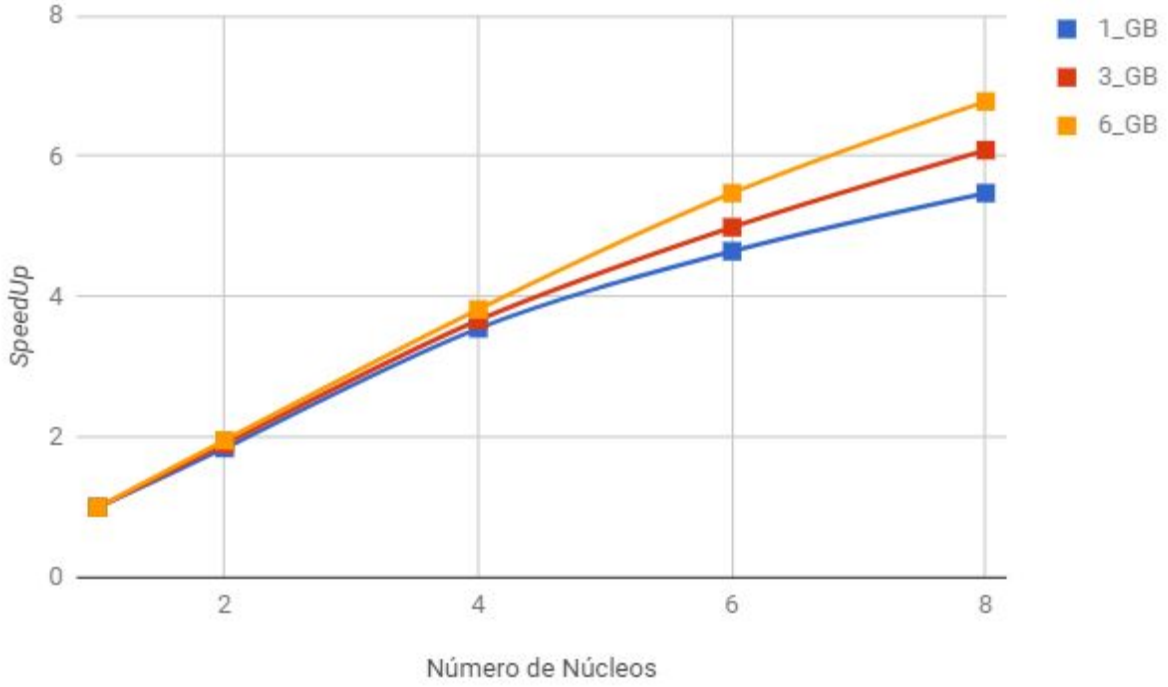
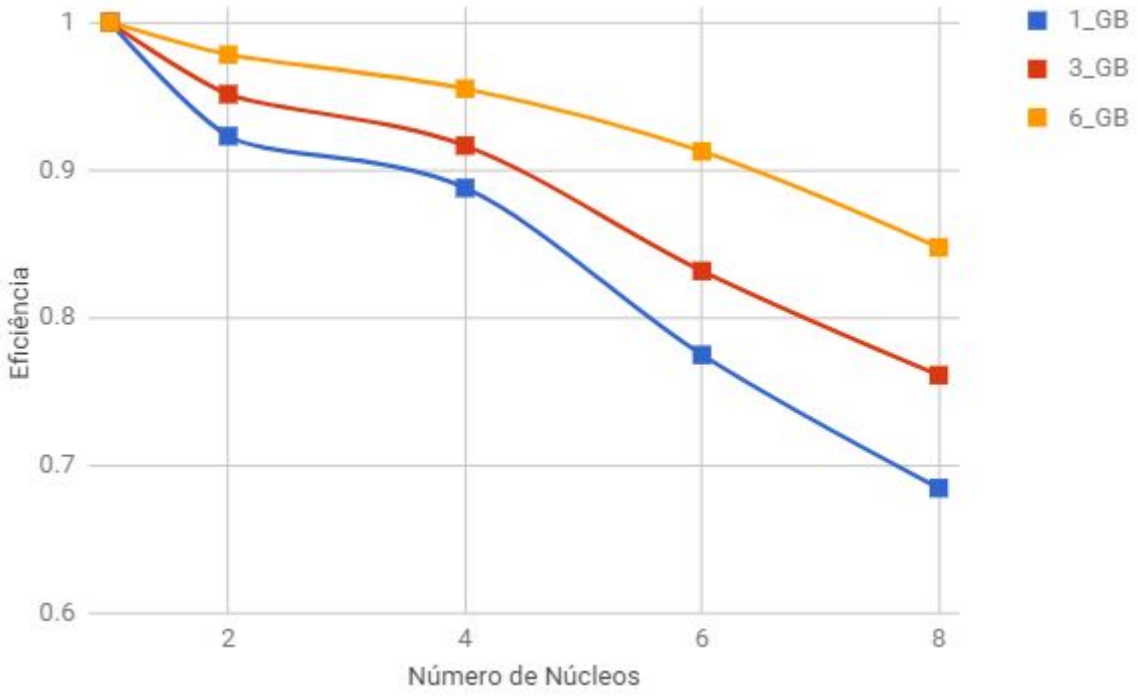


Figura 5.3 Eficiência WordCount



5.3.1.2 Vocabulário

Os tempos da etapa de vocabulário não foram colocados neste trabalho por serem muito pequenos. O processo de filtrar o vocabulário é de baixa complexidade e os tempos coletados em geral ficaram muito próximos. O vocabulário filtrado, dos arquivos de 1, 3 e 6 GB, utilizados nos testes, ficou com 160, 440 e 920 milhões de palavras.

5.3.1.3 Coocorrência

A figura 5.4 mostra o gráfico dos tempos coletados nos experimentos da geração das tuplas de coocorrência entre as palavras do texto. Apesar deste gráfico ser muito parecido com o da etapa de WordCount, os gráficos das figuras 5.5 e 5.6 nos mostram que o tamanho do arquivo não parece influenciar os resultados de *speedup* e da eficiência, pois estes valores ficaram muito próximos nos três experimentos. Isso provavelmente acontece pois nesta etapa temos que analisar cada linha, do texto de entrada, várias vezes, uma primeira vez para retirar as palavras que não fazem parte do vocabulário e depois para cada palavra presente na frase temos que verificar as coocorrências à sua direita e à sua esquerda.

Figura 5.4 Tempo de execução Coocorrência

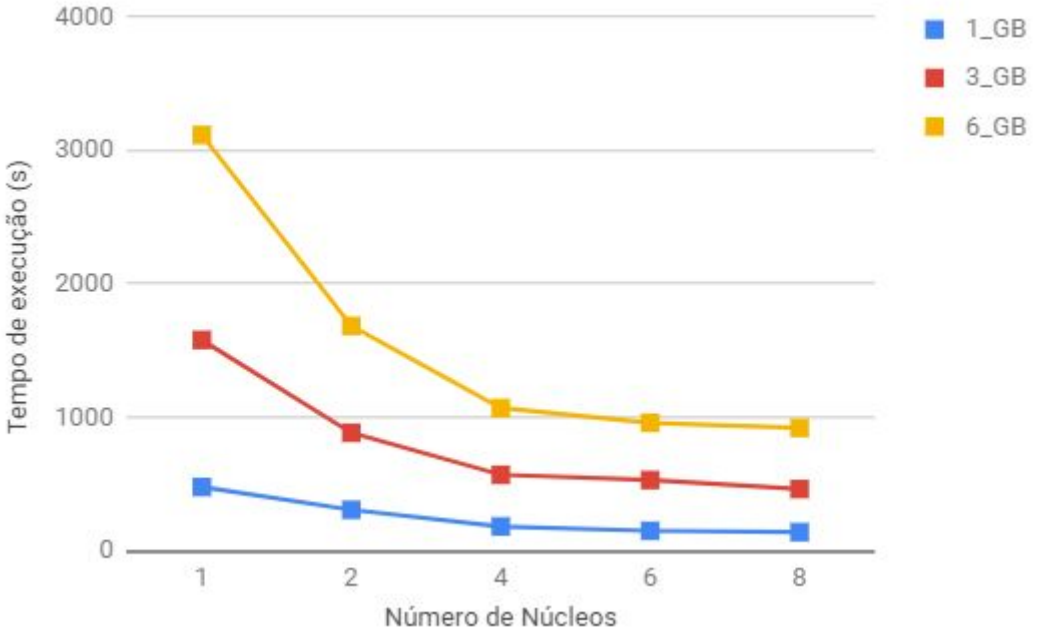


Figura 5.5 *Speedup* Coocorrência

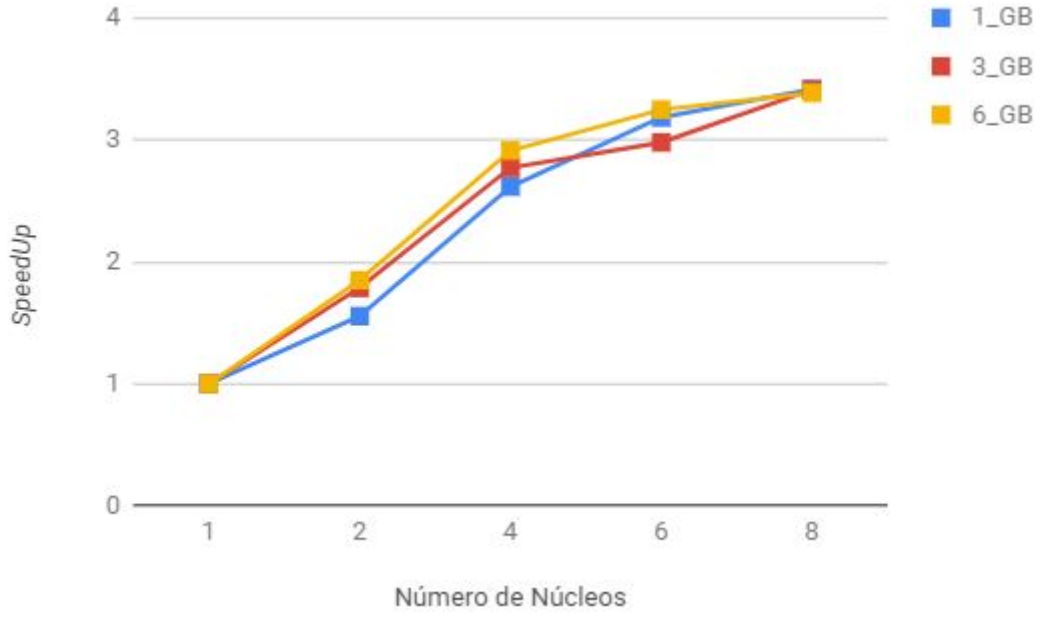
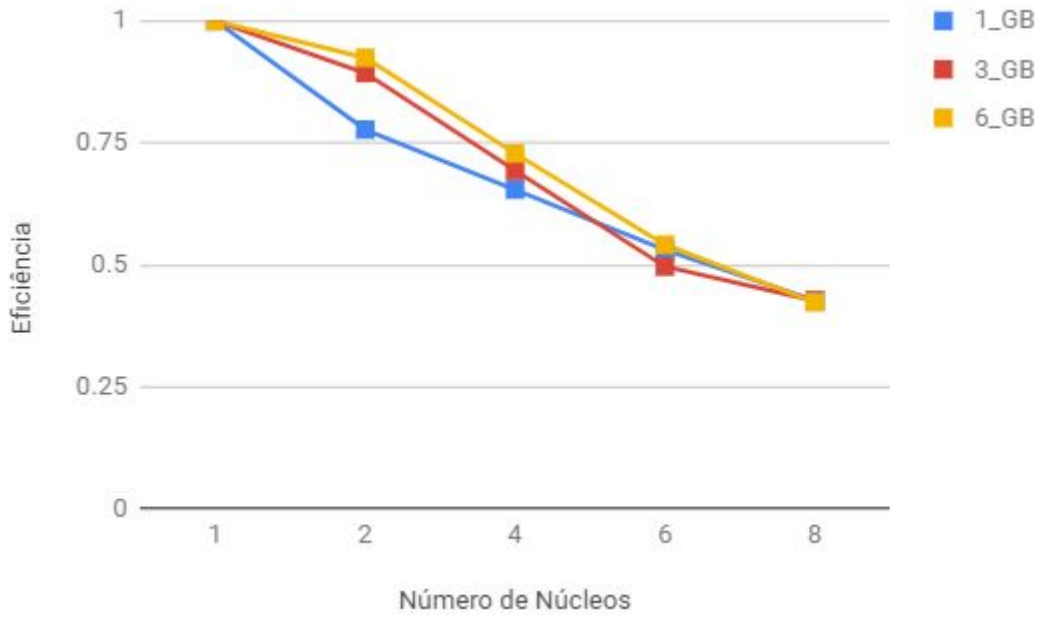


Figura 5.6 Eficiência Coocorrência



5.3.1.4 PPMI

Os tempos coletados nos experimentos desta etapa são exibidos na figura 5.7. Estes tempos mostram que a implementação desta parte da solução não conseguiu se aproveitar das vantagens do modelo de Mapreduce. Na figura 5.8 e na figura 5.9 observamos que o aumento do paralelismo não aumenta o *speedup* da etapa e que a eficiência no uso dos processadores cai muito rapidamente. A implementação da solução, nesta parte do problema, realiza apenas duas funções de mapeamento seguidas e nenhuma de redução, tornando o modelo incompleto. Além disso nesta etapa são necessários processamentos dos arquivos de entrada da etapa, que devem ser realizados sequencialmente, como a contagem de quantas coocorrências temos ao total e em quantas coocorrências cada palavra aparece individualmente. Estes valores são utilizados no cálculo do PPMI e o *Framework* não parece ter conseguido conciliar estes processamentos sequenciais com o processamento paralelo de mapeamento.

Figura 5.7 Tempo de execução PPMI

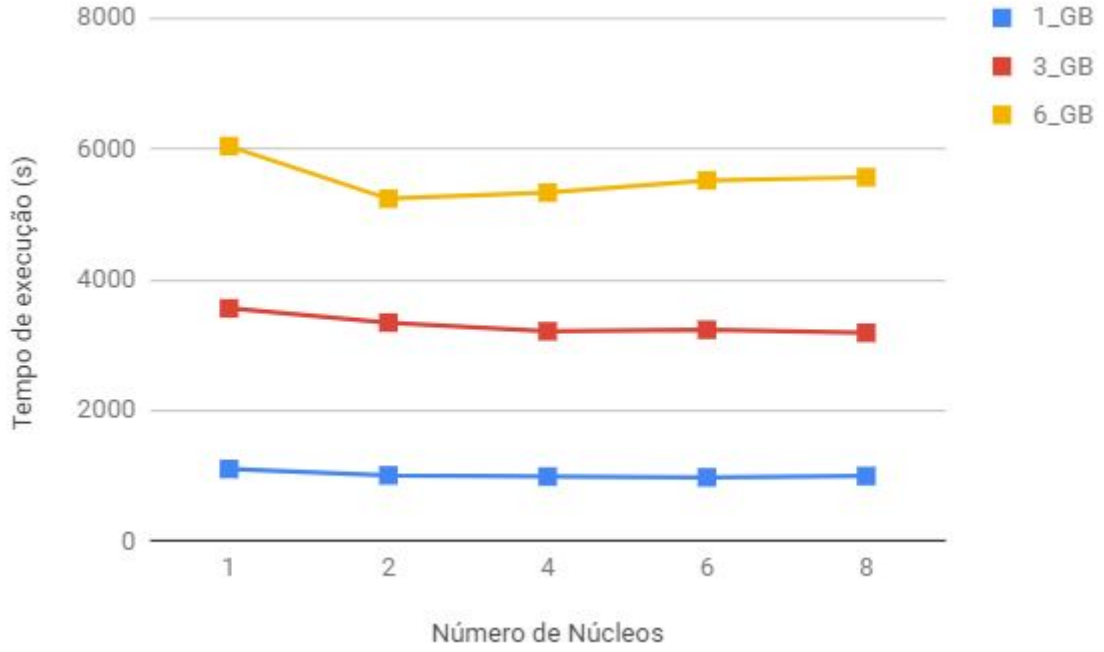


Figura 5.8 *Speedup* PPMI

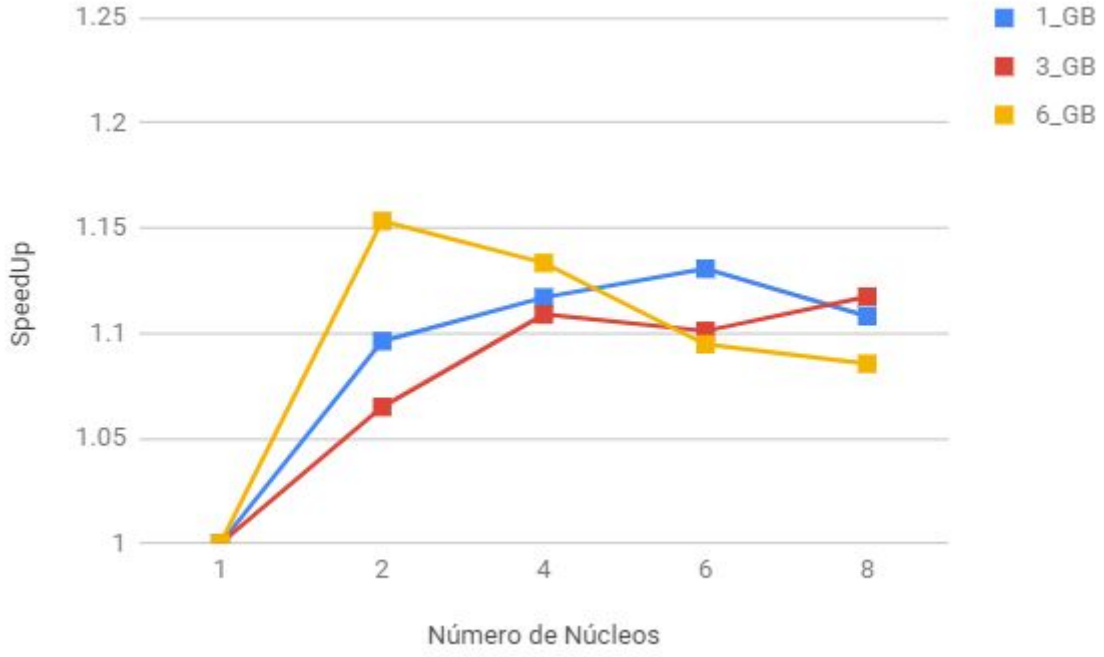
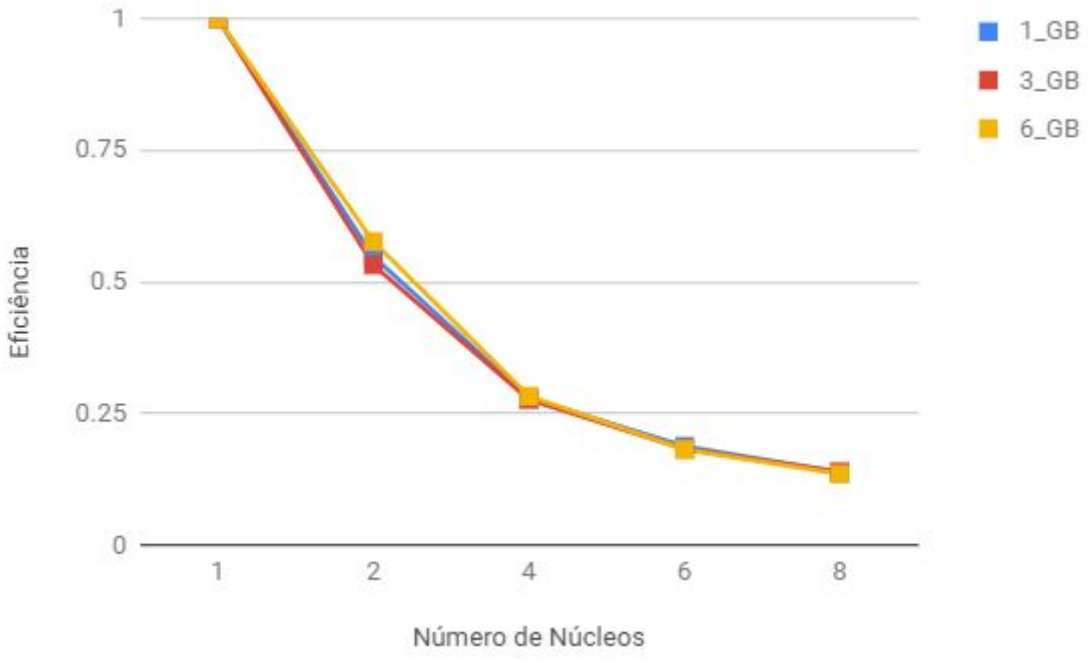


Figura 5.9 Eficiência PPMI



5.3.2 Gradiente Descendente Estocástico

Como estamos tentando aproximar vetores, gerados de forma aleatória com o gradiente descendente estocástico, é necessário executar a etapa de SGD diversas vezes. Aqui analisaremos se o método é escalável coletando o tempo para executar uma iteração completa, independente se conseguimos diminuir o erro. Foi escolhido uma divisão com 16 blocos para os vetores das palavras do vocabulário filtrado, o que cria 256 setores que diferentes.

Está é a principal etapa do método e a mais custosa, já que ela deve ser executava várias vezes para aproximarmos os vetores gerados da matrix PPMI. Os tempos coletados, de uma iteração completa, podem ser vistos na figura 5.10 e nas figuras 5.11 e 5.12 podemos ver o *speedup* e a eficiência calculadas. Pelos gráficos podemos ver que aumentando a tamanho do arquivo utilizado a eficiência parece cair mais devagar e o *speedup* crescer mais rapidamente.

Figura 5.10 Tempo de execução SGD

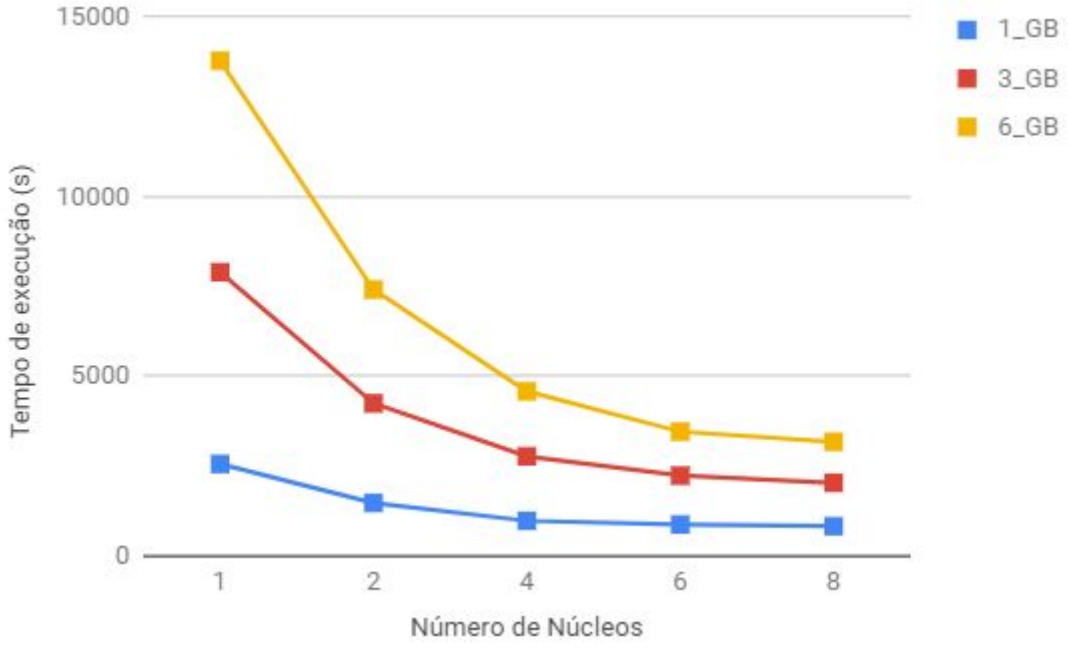


Figura 5.11 *Speedup* SGD

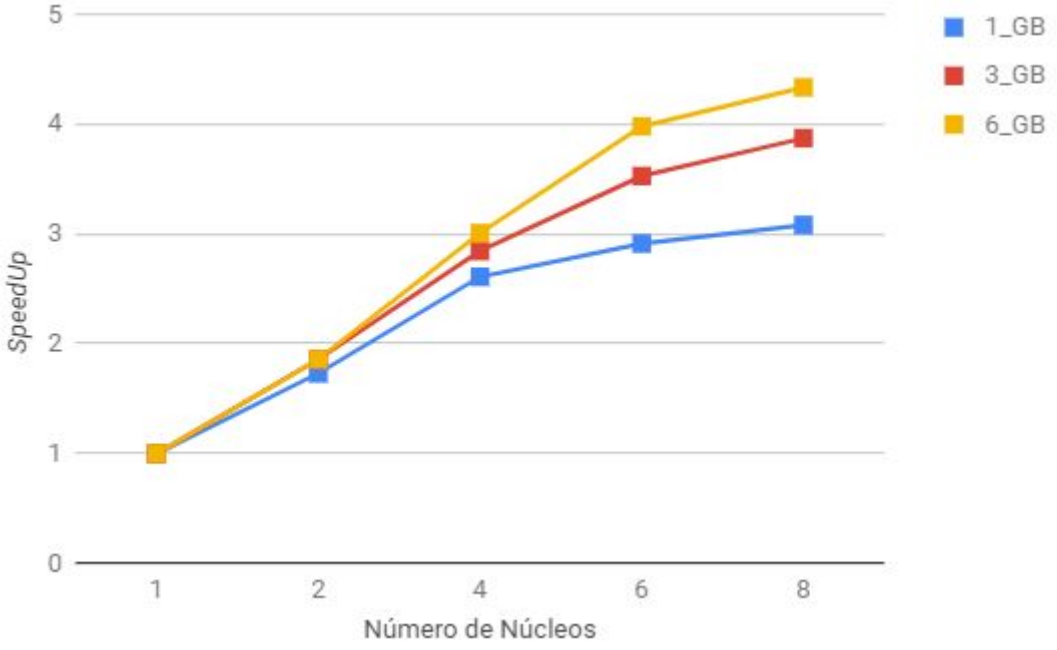
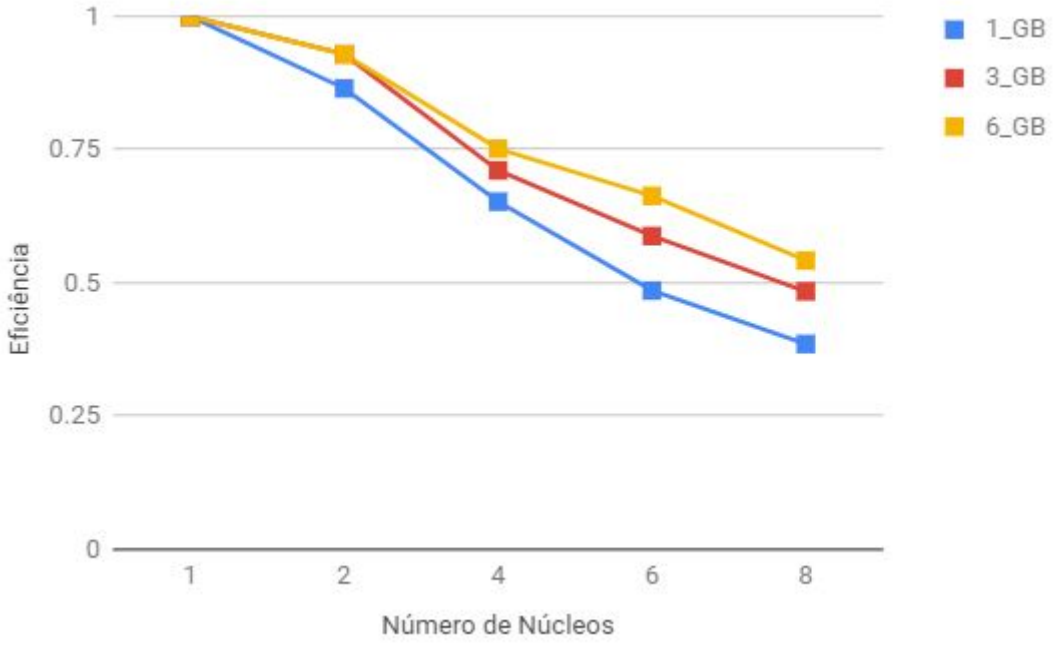


Figura 5.12 Eficiência SGD



5.4 Considerações Finais

Neste capítulo foram apresentados os resultados obtidos nos testes realizados, e uma análise de cada etapa. Apesar de quase todo o modelo ter sido implementado no modelo de *MapReduce* nem todas etapas apresentaram ganho com o aumento do paralelismo. Apesar disso a etapa de SGD, que realiza a fatoração da matriz PPMI e a mais complexa do modelo, pareceu conseguir demonstrar uma melhora nos tempos de execução e um declínio menor na curva de eficiência da utilização dos processadores com o aumento do paralelismo empregado. No próximo capítulo as conclusões do trabalho serão descritas e trabalhos futuros são comentados.

6 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho se propôs a implementar uma forma de fatorar uma matriz PPMI utilizando o gradiente descendente estocástico no modelo de programação MapReduce. Foi possível implementar não apenas a parte de fatoração pelo gradiente descendente, como também o pré-processamento, que transforma o texto que queremos processar nas entradas da etapa de SGD. Apenas a parte de unir os arquivos, que contêm as coocorrências positivas e negativas, não foi implementado no modelo de *MapReduce*. Toda a solução foi implementada na linguagem de programação Java e utilizando o *framework* Apache Flink.

A aplicação desenvolvida recebe como entrada apenas o texto para o qual queremos gerar os vetores. Durante a parte do pré-processamento contamos além das coocorrências o número de total de palavras presentes no texto. Nesta etapa é calculada a matriz PPMI de acordo com as informações que coletamos e geradas coocorrências negativas que representam coocorrências que não existem no texto. Isso auxilia na aproximação que foi proposta pois faz com que a solução leve em conta, na geração dos vetores, relações que não devem existir.

Diversos experimentos foram realizados para verificar a precisão dos resultados e o desempenho da aplicação em termos de paralelismo. Os resultados mostram que este novo método é viável em termos de ganho de tempo se empregando mais máquinas para calcularmos a solução. Apesar disso nem todas as etapas do pré-processamento mostraram ganho com o aumento do paralelismo. Os resultados da etapa da geração da matriz PPMI não demonstrou ganho com o aumento do paralelismo e a etapa de geração das ocorrências não mostrou melhorar a eficiência da utilização dos processadores com o aumento do arquivo de entrada. Muitos parâmetros que são utilizados dentro das funções destas etapas, e são calculados antes dos eventos paralelos, podem ter prejudicado os resultados finais.

A etapa da fatoração da matriz PPMI vía gradiente descendente estocástico, mais onerosa em termos de tempo de processamento e utilização de recursos da máquina, demonstrou melhora com o aumento do paralelismo utilizado, apesar de apresentar um consumo de memória muito grande. Isso fez com que o processamento completo da matriz precisasse ser dividido e seus resultados intermediários salvos em disco a cada iteração.

6.1 Trabalhos Futuros

O principal objetivo do trabalho, que era criar um método para aplicar a fatoração de matrizes pelo gradiente descendente estocástico utilizando o modelo de MapReduce, para paralelizar o processo, foi alcançado. Existem muitas decisões dentro do modelo que podem ser exploradas na procura por resultados melhores. O grande número de detalhes que a construção deste método necessita faz com que muitos fatores influenciam o resultado final. Neste trabalho utilizamos, sempre que possível, valores já referenciados em outras publicações, mas estes parâmetros podem se comportar de maneira diferente quando aplicados a este modelo de programação.

Procurar novos meios de aperfeiçoar o método podem ser explorados em novos trabalhos, focando tanto na parte de desempenho quanto na parte de precisão dos resultados. A parte do pré-processamento apresentou resultados ruins em algumas etapas do processo e pode ser melhorada, apesar de não ser a etapa principal do trabalho. Na etapa de SGD novas maneiras de conseguir realizar o processamento completo da matriz PPMI mantendo em memoria RAM as etapas intermediárias podem resultar em um ganho no desempenho da implementação.

Realizar novos experimentos aumentando o número máquinas, processadores e utilizando textos de entrada maiores também podem ser explorados em novas pesquisas. Outra possibilidade de trabalhos futuros seria aplicar o modelo a outras áreas que também utilizem técnicas de fatoração de grandes matrizes. Uma destas aŕeas são os sistemas de recomendações, muito utilizados atualmente em sites e aplicações.

REFERÊNCIAS

FLINK. Apache Flink. 2018.<flink.apache.org>. Acessado em 04 de junho, 2018.

ANJOS, J. C. S.; FEDAK, G.; GEYER, C. F. R. Bighybrid: a simulator for mapreduce applications in hybrid distributed infrastructures validated with the grid5000 experimental platforms. Concurrency and Computation: Practice and Experience, Wiley Online Library, p. 2317–2563. 2016.

ASSUNÇÃO, M. D. et al. Big data computing and clouds: Trends and future directions. Journal of Parallel and Distributed Computing, Elsevier, v. 79-80, p. 3–15, may 2015.

CHOWDHURY, G. Natural language processing. Annual Review of Information Science and Technology, 37. pp. 51-89. 2003.

CHURCH, K.W.; HANKS, P. Word association norms, mutual information, and lexicography. Computational Linguistics 16(1):22–29. 1990.

COLLOBERT, R.; WESTON, J.; BOTTOU, L.; KARLEN, M.; KAVUKCUOGLU, K.; KUKSA, P. Natural language processing (almost) from scratch.The Journal of Machine Learning Research 12:2493–2537. 2011.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. 2008.

ECKERT, C.; YOUNG, G. The approximation of one matrix by another of lower rank. Psych.1:211–218. 1936.

GEMULLA, R.; NIJKAMP, E.; HAAS, P.J.; SISMANIS, Y. Large-scale matrix factorization with distributed stochastic gradient descent. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 69–77. ACM, 2011.

IBM. 10 key marketing trends for 2017 and ideas for exceeding customer expectations. 2017.

JURAFSKY, D.; MARTIN, J. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. 2008.

KUSHNER, H. J.; YIN, G. Stochastic Approximation and Recursive Algorithms and Applications. Springer, 2nd edition, 2003.

LEE, K.-H.; LEE, Y.-J.; CHOI, H.; CHUNG, Y. D.; MOON, B. Parallel data processing with MapReduce: a survey. ACM SIGMOD Record, 40(4):11. 2012.

LEVY, O.; GOLDBERG, Y.; DAGAN, I. Improving distributional similarity with lessons learned from word embeddings.Transactions of the Association for Computational Linguistics pages 211–225. 2015.

LIDDY, E. D. Natural Language Processing. In: Drake MA, editor. Encyclopedia of library and information science. 2nd ed. New York. 2001.

LIN, D. Automatic retrieval and clustering of similar words. Inof the 36th and 17th, Volume 2. Montreal, Quebec, Canada, pages 768–774. 1998.

MANYIKA, J.; CHUI, M.; BROWN, B.; BUGHIN, J.; DOBBS, R.; ROXBURGH, C.; HUNG BYERS, A. Big data: The next frontier for innovation, competition, and productivity. 2011.

MIKOLOV, T.; CHEN, K.; CORRADO, K.; Dean, J. Efficient estimation of word representations in vector space. 2013a.

MIKOLOV, T.; YIH, W-T.; ZWEIG, G. Linguistic regularities in continuous space word representations. 2013b.

PENNINGTON, J.; SOCHER, R.; MANNING, C. D. Glove: Global vectors for word representation. Proceedings of the Empirical Methods in Natural Language Processing (EMNLP 2014)12. 2014.

REINSEL, D.; GANTZ, J.; RYDNING, J. Data age 2025: The evolution of data to life-critical. Don't focus on big data; Focus on the data that's big. Framingham: IDC Analyze the Future. 2017.

SALLE, A.; VILLAVICENCIO, A.; IDIART, M. Matrix factorization using window sampling and negative sampling for improved word representations. 2016a.

SALLE, A.; IDIART, M.; VILLAVICENCIO, A. Enhancing the lexvec distributed word representation model using positional contexts and external memory. 2016b.

SHVACHKO, K et al. The hadoop distributed file system. In: IEEE/ACM INTERNATIONAL SYMPOSIUM, 26th Symposium on Mass Storage Systems and Technologies (MSST). 2010.

TURIAN, J.; RATINOV, L.; BENGIO, Y. Word representations: a simple and general method for semi-supervised learning. InProceedings of the 48th annual meeting of the association for computational linguistics. Association for Computational Linguistics, pages 384–394. 2010.

VARELLA, A.; PEREIRA, V.; VONHELD, V.; SILVA, G. Uma interface em Linguagem Natural para a verificação de Regras de Negócio em bases de dados. 2018.

VIEIRA, R.; LIMA, V.L.S. Lingüística computacional: princípios e aplicações. In:IX Escola de Informática da SBC-Sul. Luciana Nedel (Ed.) Passo Fundo, Maringá,São José. SBC-Sul, 2001.

WHITE, T.Hadoop: The Definitive Guide. 4th. ed. [S.l.]: O'Reilly Media, 2015.