Pontifícia Universidade Católica
do Rio de Janeiro

César de Souza Bouças

Análise de dependência baseada em transição aplicada a Universal Dependencies

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informáticado da PUC-Rio.

Orientador: Prof. Ruy Luiz Milidiú

Rio de Janeiro
Outubro de 2018

Pontifícia Universidade Católica
do Rio de Janeiro

César de Souza Bouças

Análise de dependência baseada em transição aplicada a Universal Dependencies

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informáticado da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Ruy Luiz Milidiú
Orientador
Departamento de Informática – PUC-Rio

Prof. Julio Cesar Sampaio do Prado Leite
Departamento de Informática – PUC-Rio

Prof. Edward Hermann Haeusler
Departamento de Informática – PUC-Rio

Prof. Márcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 22 de Outubro de 2018

César de Souza Bouças

Bacharel em Ciência da Computação pela Universidade Federal Fluminense (Niterói, Rio de Janeiro, Brasil)

Ficha Catalográfica

Bouças, César de Souza

Análise de dependência baseada em transição aplicada a Universal Dependencies / César de Souza Bouças; orientador: Ruy Luiz Milidiú. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2018.

v., 72 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Aprendizado de máquina;. 3. Análise de dependência;. 4. NLP. I. Milidiú, Ruy Luiz. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Para minha mãe,
que descanse em paz.

Agradecimentos

Gostaria de agradecer a todos os integrantes do Departamento de Informática da PUC-Rio. Em especial aos professores: Ruy Milidiú pela orientação e suporte, Sérgio Lifschitz pela oportunidade na monitoria, Clarisse Sieckenius de Souza pelas aulas inspiradoras, Júlio Leite e Edward Hermann pela participação na banca. E aos coordenadores Alberto Raposo e Eduardo Laber. Agradeço também à Regina Zanon e Cosme Pereira Leal, sempre cordiais no atendimento da secretaria.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Resumo

Bouças, César de Souza; Milidiú, Ruy Luiz. **Análise de dependência baseada em transição aplicada a Universal Dependencies**. Rio de Janeiro, 2018. 72p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Análise de dependência consiste em obter uma estrutura sintática correspondente a determinado texto da linguagem natural. Tal estrutura, usualmente uma árvore de dependência, representa relações hierárquicas entre palavras. Representação computacionalmente eficiente que vem sendo utilizada para lidar com desafios que surgem com o crescente volume de informação textual online. Podendo ser utilizada, por exemplo, para inferir computacionalmente o significado de palavras das mais diversas línguas. Este trabalho apresenta a análise de dependência com enfoque em uma de suas modelagens mais populares em aprendizado de máquina: o método baseado em transição. Desenvolvemos uma implementação gulosa deste modelo com um classificador neural simples para executar experimentos. Datasets da iniciativa *Universal Dependencies* são utilizados para treinar e posteriormente testar o sistema com a validação disponibilizada na tarefa compartilhada da CoNLL-2017. Os resultados mostram empiricamente que se pode obter ganho de performance inicializando a camada de entrada da rede neural com uma representação de palavras obtida com pré-treino. Chegando a uma performance de 84,51 LAS no conjunto de teste da língua portuguesa do Brasil e 75,19 LAS no conjunto da língua inglesa. Ficando cerca de 4 pontos atrás da performance do melhor resultado para analisadores de dependência baseados em sistemas de transição.

Palavras-chave

Aprendizado de máquina; Análise de dependência; NLP

Abstract

Bouças, César de Souza; Milidiú, Ruy Luiz (Advisor). **Transition-based dependency parsing applied on Universal Dependencies**. Rio de Janeiro, 2018. 72p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Dependency parsing is the task that transforms a sentence into a syntactic structure, usually a dependency tree, that represents relations between words. This representations are useful to deal with several tasks that arises with the increasing volume of textual online information and the need for technologies that depends on NLP tasks to work. It can be used, for example, to enable computers to infer the meaning of words of multiple natural languages. This paper presents dependency parsing with focus on one of its most popular modeling in machine learning: the transition-based method. A greedy implementation of this model with a simple neural network-based classifier is used to perform experiments. Universal Dependencies treebanks are used to train and then test the system using the validation script published in the CoNLL-2017 shared task. The results empirically indicate the benefits of initializing the input layer of the network with word embeddings obtained through pre-training. It reached 84.51 LAS in the Portuguese of Brazil test set and 75.19 LAS in the English test set. This result is nearly 4 points behind the performance of the best results of transition-based parsers.

Keywords

Machine learning; Dependency parsing; NLP

Sumário

1
Introdução

Processamento da Linguagem Natural (NLP) é uma área de pesquisa que explora como sistemas computacionais podem ser criados e utilizados para entender e manipular o texto e a fala da linguagem natural. Dentre os tópicos abordadas em NLP, existe a tarefa de determinar automaticamente uma estrutura sintática correspondente a uma dada sentença escrita em linguagem natural. Esta tarefa é chamada de análise de dependência (*Dependency Parsing*) quando essa estrutura sintática é representada através de uma gramática de dependência.

A figura 1.1 ilustra o problema de análise de dependência mostrando um exemplo de entrada e saída correspondente.

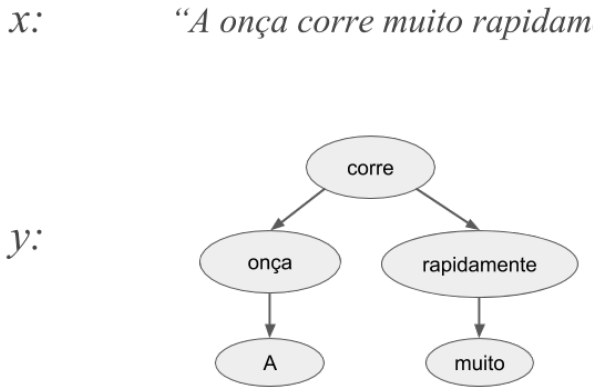


Figura 1.1: A entrada da análise de dependência é uma sentença, denotada por x na ilustração. A saída, aqui representada pela letra y, é a árvore sintática correspondente à sentença de entrada. No aprendizado de máquina, o objetivo é obter uma função f tal que $f(x) = y$.

Na prática uma árvore de dependência carrega mais informações, como rótulos nas ligações entre seus elementos e tags POS. Ela geralmente é representada como na figura 1.2. Que ilustra uma árvore de dependência correspondente a seguinte frase escrita em língua portuguesa: *"O clima geral é de resignação."*, extraída do conjunto *Universal Dependencies*.

Lidar computacionalmente com linguagem natural é uma tarefa desafiadora. Parte devido a existência de uma certa ambiguidade, que é intrínseca

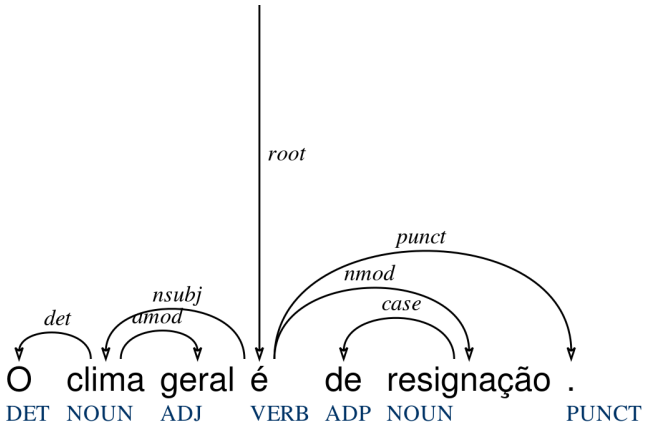


Figura 1.2: Estrutura (árvore) de dependência com arcos de dependência rotulados e tags POS.

às linguagens naturais: uma mesma palavra pode assumir diferentes significados dependendo do contexto. Parte devido ao fato de que a linguagem natural geralmente assume que seu receptor possui conhecimento prévio sobre o contexto da mensagem, sobre o mundo ou sobre o senso comum. Tome por exemplo a seguinte sentença da língua portuguesa:

"Olá, encontrei o treinador e resolvemos fazer uma reunião em sua sala".

Neste caso, poderia surgir dúvida sobre a sala na qual será a reunião. Seria um sistema computacional capaz de interpretar corretamente a ambiguidade?

Análise de dependência ajuda a lidar com este desafio proporcionando uma estrutura sintática que auxilia no processamento computacional da linguagem natural. Estando presente em diversas aplicações de NLP, como tradutores e extratores de informação.

Outro desafio diz respeito à diversidade quantitativa e qualitativa das linguagens naturais existentes no mundo. Que são da ordem de milhares, cada qual com sua própria gramática. Imagine construir e manter um sistema específico para lidar com cada uma delas. Além disso, as línguas naturais evoluem com o tempo e passam por reformas ortográficas, fazendo necessárias atualizações de sistemas computacionais que lidam com elas.

As estratégias mais bem sucedidas atualmente para lidar com esse problema utilizam técnicas de aprendizado de máquina. A utilização dessas técnicas se beneficia de grandes volumes de dados amostrais. Possibilitando que se obtenha um analisador sintático (*parser*), a partir de um dataset especial. Que é formado por um conjunto de dados textuais sintaticamente anotado, conhecido como corpus, ou banco de árvores (*treebank*).

No entanto, tal abordagem só é possível quando os dados necessários para treinar e ajustar o sistema estão disponíveis. E este conjunto de dados anotados é relativamente caro de se produzir. No caso da análise de dependência, esses dados vem se tornando cada vez mais disponíveis graças a iniciativas como *Universal Dependencies* (UD). Que propõe uma forma de anotação sintática consistente entre línguas diversas, e que disponibiliza bancos de árvores de livre acesso ao público para mais de 60 línguas. Isso, juntamente com o fato de a estrutura sintática produzida pelo parser ser computacionalmente útil e barata, faz aumentar o interesse da comunidade em análise de dependência. Que vem se consolidando nos últimos anos como uma tarefa importante no contexto das aplicações de NLP.

Neste trabalho abordamos uma das modelagens de sistemas de análise de dependência mais populares atualmente: análise baseada em transição. Apresentando suas principais características e mostrando seu funcionamento na prática, através de uma implementação gulosa do método, baseada no trabalho de Chen e Manning 2014.

Realizamos experimentos utilizando datasets UD das línguas portuguesa e inglesa. Eles mostram que esta abordagem apesar de simples, pode apresentar uma precisão satisfatória quando comparada ao melhor resultado obtido por um sistema baseado em transição. Tais experimentos foram conduzidos utilizando o mesmo setup utilizado pela maioria dos sistemas participantes da tarefa compartilhada na CoNLL-2017. A Figura 1.3 ilustra este setup.

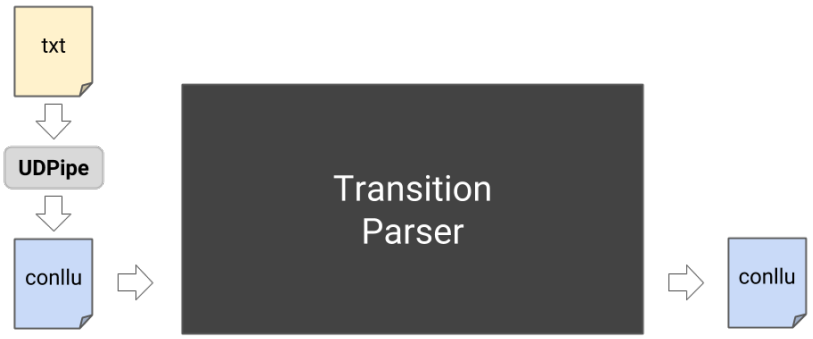


Figura 1.3: O arquivo de entrada em texto puro é processado utilizando UDPipe para realizar as tarefas preliminares necessárias. Gerando um arquivo no formato Universal Dependencies, que é processado pelo nosso analisador baseado em transição. A saída é então anotada em um arquivo com o mesmo formato para posterior avaliação.

Utilizamos arquivos processados com *UDPipe* (Straka e Straková 2017) como entrada para nosso analisador. Estes arquivos contém as sentenças preparadas para realização da análise de dependência. Isto é, tarefas de

segmentação, tokenização e tags POS anotadas pelo *UDPipe*. Portanto um arquivo anotado com *silver tags*.

Esta estratégia permite comparação direta da performance dos analisadores obtidos com este trabalho com a performance dos sistemas participantes na conferência.

1.1 Contextualização

Na modelagem das aplicações do processamento de linguagem natural, várias tarefas são realizadas em sequência, a fim de tornar um texto pronto para ser tratado pelo analisador sintático. Tal como segmentação e previsão de rótulos part-of-speech (POS), que representam uma análise morfológica do texto. Além disso, a saída do parser é usada como entrada para outras tarefas comuns de NLP, como por exemplo tarefas que envolvem interpretação semântica. Assim, dizemos que dependency parsing faz parte de um fluxo de tarefas, conhecido como Pipeline NLP.

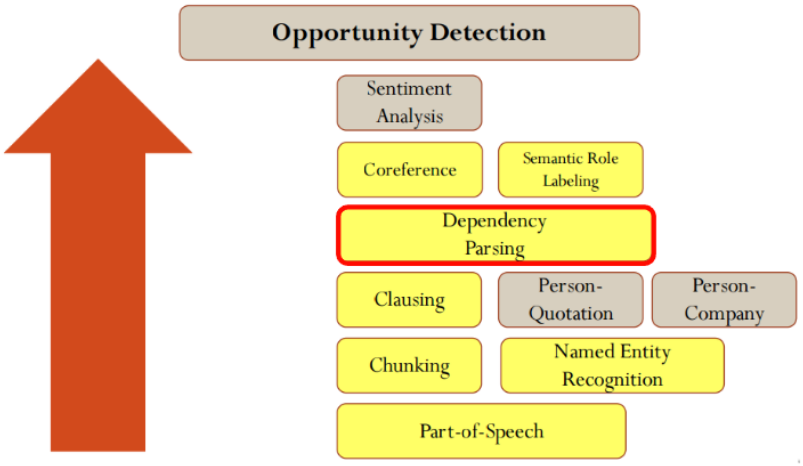


Figura 1.4: Uma representação de parte do pipeline NLP extraída das notas de aula do professor Ruy Milidíu. Cada caixa representa uma tarefa e a seta indica o fluxo de execução. Quanto mais próximo do texto original é a entrada da tarefa mais embaixo na pilha ela se encontra.

Existem diversas abordagens utilizadas para se construir analisadores sintáticos. A abordagem apresentada neste trabalho é classificada como sendo orientada aos dados, *Data-driven dependency parsing*. Pois faz uso essencial de técnicas de aprendizado de máquina, para, a partir de um dataset, aprender um analisador (parser) capaz de analisar novas sentenças da linguagem natural presente no dataset.

Esta abordagem necessita portanto de um conjunto de dados sintaticamente anotado, de forma que cada sentença escrita em linguagem natural esteja associada a sua estrutura sintática correspondente. Trata-se portanto de um problema de aprendizado de máquina supervisionado.

Embora o foco deste trabalho esteja em métodos supervisionados, vale lembrar que também há uma linha de trabalho considerável na utilização de métodos não supervisionados. Estes métodos não necessitam de conjuntos de dados sintaticamente anotados, mas apresentam performances inferiores às abordagens supervisionadas.

Dado que a saída de um parser é uma estrutura sintática, isso classifica o problema como sendo um problema de aprendizado estruturado.

Análise de dependência tem sido cada vez mais utilizada em diversas aplicações do processamento de linguagem natural. Por exemplo: extração de informação (Culotta e Sorensen 2004), tradutores (*Machine Translation*) (Ding e Palmer 2004) e resposta automatica (*Question Answering*) (Wang, Smith e Mitamura 2007).

1.2 Arquitetura

Existem duas abordagens principais quando se trata de análise de dependência orientada aos dados: uma baseada em grafo (R. T. McDonald, F. Pereira et al. 2005) e outra baseada em transição. Esta última foi proposta por Yamada e Matsumoto 2003 e por Nivre 2006.

Dada uma sentença de entrada escrita em linguagem natural, o algoritmo baseado em grafo encontra a árvore de dependência de pontuação mais alta dentre todas as saídas possíveis, analisando o grafo completo das relações de dependência.

Alternativamente, um algoritmo baseado em transição cria a árvore através de uma sequencia de ações, analisando cada ação individualmente de forma sequencial. Seu funcionamento se assemelha ao de um autômato. A Figura 1.5 apresenta uma ilustração da arquitetura de um sistema baseado em transição no processamento de uma sentença da língua portuguesa.

Esta arquitetura é atraente pela sua eficiência e por permitir adaptar o sistema para analisar sentenças de línguas diferentes mantendo a mesma estrutura. De fato, a análise multi-lingue é uma motivação para esta abordagem. Que utiliza técnicas de aprendizado de máquina para aprender classificadores de diversas línguas e os incorpora ao sistema.

O funcionamento de cada componente desta arquitetura será explicado em mais detalhes no decorrer deste trabalho.

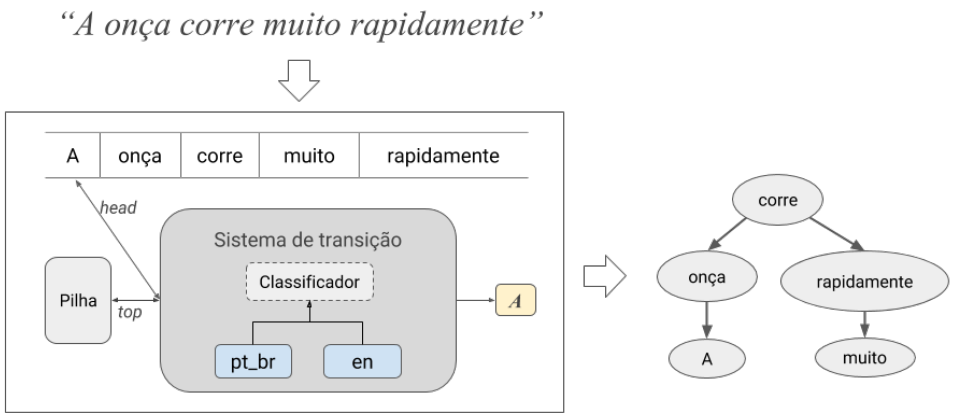


Figura 1.5: O sistema recebe como entrada uma sentença e produz como saída a árvore de dependência correspondente. Em um dado momento do processamento, o sistema apresenta configuração c_i. Que se refere ao estado de suas estruturas internas: o buffer, a pilha e o conjunto de arcos de dependência derivados A. Dizemos que A é a árvore parcial da análise em um dado momento.

1.3 Histórico

A abordagem de análise de dependência orientada a dados foi originalmente proposta por Eisner 1996 utilizando-se um método de análise baseado em grafo. Na ocasião ainda não existia a terminologia *graph-based*, que só veio a aparecer cerca de dez anos depois.

Foi com o trabalho de McDonald e Nivre (2007) que os termos *Graph-based* e *Transition-based* apareceram. Eles foram utilizados para diferenciar o sistema *MSTParser* (R. T. McDonald e F. C. N. Pereira 2006; R. T. McDonald, F. Pereira et al. 2005) do *MaltParser* (Nivre 2006), respectivamente. Vale lembrar que a mesma distinção foi proposta anteriormente por Buchholz e Marsi 2006, porém utilizando outra terminologia: *All pairs* e *Stepwise* para designar os sistemas baseados em grafo e em transição respectivamente.

As duas metodologias alcançaram performances semelhantes na tarefa compartilhada na conferência CoNLL-X em 2006 através das implementações *MSTParser* e *MaltParser*. Que até hoje são consideradas como referência de implementações destes dois métodos. O *MSTParser* é uma implementação do método baseado em grafo, enquanto o *MaltParser* implementa o método baseado em transição.

A primeira aparição do método baseado em transição se deu através de dois trabalhos que propuseram uma implementação gulosa utilizando *Support Vector Machines* como classificador: Yamada e Matsumoto 2003 e Nivre 2003. Sendo este último, o trabalho precursor do que é o sistema *MaltParser*

atualmente.

As primeiras implementações dos modelos baseados em transição foram implementações gulosas, no sentido de que o sistema guloso sempre seleciona a transição melhor ranqueada por um classificador, ignorando outras alternativas. Logo percebeu-se que apesar de eficiente e simples de implementar, o método guloso sofria com o problema da propagação de erros. Que advém da incapacidade do sistema de se recuperar de decisões incorretas. Ou seja, se em um dado momento analisando uma sentença um erro for cometido, isso fará com que toda a estrutura sintática gerada a partir deste momento sofra com a propagação do erro. O que acaba refletindo negativamente na performance (precisão, acurácia, qualidade) do sistema.

Diversos trabalhos mostraram como técnicas que amenizam este problema podem melhorar a performance de analisadores sintáticos. De fato, analisadores baseados em transição que ao invés da estratégia gulosa utilizam buscas globais (R. T. McDonald, F. Pereira et al. 2005, Koo e Collins 2010) ou heurísticas de busca como busca em feixe (*beam-search*) (Zhang e Clark 2008) ou ainda analisadores que utilizam técnicas de programação dinâmica (Huang e Sagae 2010; Kuhlmann, Gómez-Rodrıguez e Satta 2011) superam implementações gulosas em performance.

Seguindo a linha de pesquisa que estuda o método baseado em transição, acompanhou-se uma extensa literatura na direção de melhorias de performance. Para alcançar tais melhorias, o esforço teve foco em um dos componente principais dos sistemas deste tipo: o classificador. Passando pela utilização de variantes do algoritmo perceptron (Zhang e Nivre 2011) até chegar a utilização das redes neurais. Que representou um ganho considerável de performance.

Assim como diversas tarefas NLP, a análise de dependência passou a se beneficiar da representação da entrada obtida através da técnica de incorporação de palavras (word embeddings). Esta técnica permite extrair uma estrutura que carrega informação semântica a partir das palavras de um texto de entrada, obtendo assim uma representação contínua das palavras. Que se popularizou com a implementação do Word2vec (Mikolov et al. 2013): uma ferramenta capaz de produzir essa representação das palavras de forma eficiente e sem a necessidade de um conjunto de treino anotado. Possuindo tamanha eficiência que permite fazer um pré-treino em conjuntos com bilhões de palavras e inicializar a camada de entrada de um classificador neural com os vetores obtidos.

Isso representava uma oportunidade de abordar um problema comum nos modelos de NLP: aliviar a escassez de atributos, fazendo o uso de ligações estatísticas entre palavras semelhantes sem a necessidade de se ter um conjunto

de treino anotado para isso. De fato esta técnica obteve grande sucesso atuando em conjunto com modelos baseados em redes neurais. Permitindo inicializar a primeira camada da rede com conhecimento sobre a relação entre palavras. Vindo a se tornar uma parte fundamental dos sistemas do estado-da-arte em análise de dependência e se consolidando como uma das aplicações do aprendizado não-supervisionado mais proeminentes.

A primeira tentativa de se utilizar a representação distribuída das palavras e de tags POS obtidas com *Word2Vec* pré treinado em um conjunto maior para inicializar a camada de entrada de um classificador neural vem com o trabalho de Chen e Manning 2014.

Este trabalho mostrou empiricamente os benefícios que esta técnica traz quando combinada com o classificador neural. Que mesmo tendo utilizado uma implementação simples e gulosa de um sistema baseado em transição, melhorou em cerca de dois pontos percentuais a performance do estado-da-arte. Como consequência diversos trabalhos exploraram formas de se conseguir algum ganho de performance partindo da mesma modelagem. Por exemplo, Weiss et al. 2015 ilustra o ganho de precisão que se pode atingir adicionando uma heurística de busca em feixe ao invés de utilizar o método guloso. Utiliza também um classificador neural com arquitetura mais profunda e um conjunto de treino aumentado. Conseguindo um ganho de precisão em torno de 2 pontos em relação a Chen e Manning 2014.

Apesar de utilizarem redes neurais, essas modelagens ainda necessitavam da definição manual de um conjunto de atributos para representar o estado do sistema de transição. Por exemplo, a representação em Chen e Manning 2014 é uma concatenação de 18 vetores de palavras, 18 vetores de tags POS e 12 vetores de rótulo de dependência.

Uma linha de trabalho diferente visa eliminar esta engenharia de atributos, sugerindo novas arquiteturas de rede neural para codificar a configuração do analisador (Dyer et al. 2015, Kiperwasser e Goldberg 2016).

Assim como em outras tarefas em aprendizado de máquina, a análise de dependência também pode se beneficiar de estratégias que utilizam engenharia de software para combinar modelos. Formando assim comitês de modelos (*Ensemble*). Combinar a abordagem baseada em grafos com a abordagem baseada em transição a fim de ter o melhor dos dois é uma ideia atraente. Embora difícil de realizar, uma vez que esses métodos dependem de dois escopos de dados completamente diferentes. Mas é possível e pode produzir bons resultados, como mostrado por Nivre e R. McDonald 2008. Assim como o trabalho de Bohnet e Kuhn 2012, que também explora diferentes formas de combinar sistemas baseados em grafo com sistemas baseados em transição.

1.4 Estado-da-arte

Dos resultados publicados nos mesmos conjuntos Universal Dependencies (UD) utilizados neste trabalho, o melhor que se tem notícia é de Stanford na CoNLL-2017. Mas este resultado advém de um sistema baseado em grafos.

Sistema	Estratégia	pt_br(LAS)	en(LAS)
Stanford	grafo	**91,36**	**82,23**
HIT-SCIR	transição	88,71	79,94

Tabela 1.1: Estado-da-arte nos conjuntos *pt-br* e *en* do Universal Dependencies 2.0.

Para a maioria dos conjuntos UD, os resultados da CoNLL representam um novo estado da arte para análise de dependência. E na tarefa compartilhada na edição de 2018 o sistema HIT-SCIR obteve a melhor performance, superando Stanford. Mas como em 2018 os datasets diferem dos utilizados neste trabalho, não iremos considerar tais resultados aqui.

Dentre os sistemas baseados em transição treinados e testados nos conjuntos UD *pt-br* e *en*, o melhor resultado é o do sistema HIT-SCIR. Publicado em Che et al. 2017 na tarefa compartilhada da CoNLL-2017. A Tabela 1.1 ilustra os melhores resultados obtidos por cada uma das abordagens: grafo e transição. Estes resultados estão publicados na CoNLL-2017.

A modelagem utilizada no HIT-SCIR utiliza redes neurais não apenas na representação da linguagem, mas para representar também cada um dos componentes que formam um sistema de transição. Para isso utiliza variações de uma rede neural recorrente com unidades de memória de curto prazo chamadas de stack-LSTMs. Onde três destas redes são utilizadas para obter incrementalmente as representações do buffer, pilha e da seqüência de ações de transição.

1.5 Conference on Computational Natural Language Learning (CoNLL)

CoNLL é uma conferência de prestígio mundial organizada anualmente pelo Grupo de Interesse Especial sobre Aprendizagem de Linguagem Natural (SIGNLL) vinculado a *Association for Computational Linguistics* - ACL. Aborda diversos tópicos de aprendizado de línguas. Destacando-se o desenvolvimento de métodos de aprendizado de máquina aplicados a tarefas de processamento de linguagem natural. Tal como processamento de fala, fonologia, morfologia, sintaxe, semântica, processamento de discurso, aplicações de engenharia de linguagem, entre outras.

Desde 1999, a CoNLL inclui uma tarefa compartilhada na qual os dados de treinamento e teste são fornecidos pelos organizadores. Isso permite que os sistemas participantes sejam avaliados e comparados de maneira sistemática. As descrições dos sistemas participantes e uma avaliação de seus desempenhos são apresentadas na conferência. Das edições[1] que tiveram tarefas relacionadas ao tema da análise de dependência podemos elencar:

- 2005: *Dependency Parsing*
- 2006: *Multilingual Dependency Parsing*
- 2007: *Dependency Parsing: Multilingual & Domain Adaptation*
- 2008: *Joint Parsing of Syntactic and Semantic Dependencies*
- 2009: *Syntactic and Semantic Dependencies in Multiple Languages*
- 2017: *Multilingual Parsing from Raw Text to Universal Dependencies*
- 2018: *Multilingual Parsing from Raw Text to Universal Dependencies*

Em especial, as edições de 2017 e 2018 incluíram uma tarefa compartilhada que aborda análise de dependência aplicada a *Universal Dependencies*. Em que a tarefa consistia em aprender um parser capaz de efetuar a análise de dependência de conjuntos de teste a partir do texto puro ou do texto com segmentação e tags produzidas pelo sistema baseline.[2]

Na edição de 2017 o sistema vencedor (Stanford) utilizou um modelo baseado em grafos. O sistema baseado em transição HIT-SCIR obteve a quarta colocação nesta ocasião. Os resultados também mostram a consolidação da técnica de incorporação de palavras (word embeddings): os poucos sistemas que não utilizaram essa técnica figuram entre as piores performances reportadas. Uma análise mais completa destes resultados pode ser encontrada em Nivre et al. 2017.

[1] http://www.conll.org/previous-tasks
[2] http://universaldependencies.org/conll17/

1.6 Motivação e objetivos

O método baseado em transição está presente em várias implementações de analisadores com performances comparáveis ao estado-da-arte. Figurando entre alguns dos sistemas de melhor desempenho na tarefa de dependency parsing das conferências CoNLL. O que revela a importância de estudar e reproduzir um sistema deste tipo, para assim entender melhor seu funcionamento.

Este trabalho tem como objetivo principal ampliar o conhecimento e proporcionar material introdutório escrito em língua portuguesa para aqueles que buscam aprender sobre o método baseado em transição. Na prática, o enfoque é reproduzir o trabalho de Chen e Manning 2014 com recursos disponibilizados na CoNLL-2017.

A tarefa de análise de dependência é aqui utilizada para mostrar como técnicas de aprendizado de máquina têm sido empregadas para lidar com os desafios intrínsecos ao processamento de linguagem natural. Mais especificamente responder a seguinte questão: como os problemas relacionados à tarefa de análise de dependência são abordados na arquitetura baseada em transição?

Com enfoque na específica abordagem baseada em transição, apresentamos o desenvolvimento teórico e prático da análise de dependência. Mostrando desde a etapa de preparação dos dados, passando pela representação das palavras e as partes que compõem um sistema deste tipo. Uma implementação de um sistema que reproduz o trabalho de Chen e Manning 2014 é utilizada para efetuar experimentos. Mostrando os benefícios da utilização de técnicas de *Deep learning* na tarefa de análise de dependência.

Este trabalho utiliza apenas recursos disponíveis ao público de forma gratuita e livre de direitos autorais. Como os datasets UD e ferramentas treináveis de análise de dependência disponibilizadas publicamente, tal qual a implementação do pipeline NLP *UDPipe*. Que foi utilizado pela maioria dos participantes da tarefa sobre análise de dependência compartilhada na edição de 2017 da CoNLL, inclusive pelos vencedores. Como também a ferramenta de pré treino de representação *Word2Vec*, utilizada para obter vetores de palavras previamente ao treino.

1.7 Contribuições

Como contribuições deste trabalho podemos elencar:

– Compilação de conceitos que permeiam a analise de dependência baseada

em transição. Sendo de interesse àqueles que desejam uma introdução escrita em língua portuguesa sobre o tema.

– Implementação de uma arquitetura baseada no trabalho de Chen e Manning 2014, que representa um bom ponto de partida para consolidar os conceitos teóricos apresentados. Permitindo posterior evolução no sentido de modelagens mais sofisticadas.

– Analisadores para as línguas portuguesa e inglesa. Treinados em conjuntos Universal Dependencies. Que alcançam performances razoável considerando a simplicidade do modelo. A Tabela 1.2 sumariza a performance obtida nos experimentos comparadas com o sistema HIT-SCIR.

Sistema	**en(LAS)**	**pt-br(LAS)**
HIT-SCIR	**79,94**	**88,71**
Este trabalho	75,19	84,51

Tabela 1.2: Métricas LAS obtidas utilizando script de validação disponibilizado na CoNLL-20017. Comparamos a performance do sistema desenvolvido neste trabalho com o melhor resultado de um sistema baseado em transição nos conjuntos utilizados.

1.8 Organização deste trabalho

Uma visão geral da arquitetura e introdução conceitual informal aos elementos que compõe a tarefa, bem como um apanhado histórico, foram apresentadas neste primeiro capítulo.

Os dois capítulos subsequentes fornecem uma caracterização do problema de análise de dependência, explorando uma classe principal de modelagem que está em uso atualmente: o método baseado em transição. Continua com um capítulo que descreve uma implementação deste método reportando também sua avaliação empírica. Fechando com algumas palavras sobre as conclusões, tendências atuais e perspectivas futuras.

Para tornar mais fácil o entendimento, algumas definições apresentadas ao longo deste trabalho omitem o rótulo das relações de dependência. Mas são análogas às definições da análise de dependência rotulada.

2
Análise de dependência

Este capítulo apresenta algumas definições para conceitos fundamentais que permeiam a análise de dependência. Incluindo a introdução da notação que será utilizada e a definição dos problemas que envolvem o método orientado aos dados: e etapa de aprendizado e a etapa de análise de novas sentenças. Também são abordados assuntos como estruturas sintáticas, representação da entrada e *Universal Dependencies* (UD).

2.1
Definição

A análise de dependência é a tarefa de mapear uma sentença de linguagem natural em uma representação de sua sintaxe ou semântica na forma de uma árvore de dependência. Temos então um problema em que:

- A entrada x é uma sentença escrita em linguagem natural.
- A saída y é uma estrutura de dependência que representa a análise sintática da entrada.

Nos termos do aprendizado de máquina, isso significa derivar um programa f a partir de um conjunto de pares anotados do tipo (sentença, estrutura) ou (x, y). Para então ser capaz de utilizar f para analisar novas entradas: $f(x') = y'$.

2.2
O método orientado aos dados: data-driven dependency parsing

Em processamento de linguagem natural, uma abordagem é dita orientada aos dados se para analisar novas sentenças ela faz uso essencial de aprendizado de máquina a partir de conjuntos de dados linguísticos. Um programa, leia-se analisador sintático, derivado dos dados através do uso de técnicas de aprendizado de máquina é chamado de modelo.

Dentre as vantagens de se utilizar abordagens orientadas aos dados em NLP temos o fato de que o tempo de desenvolvimento pode se tornar muito menor quando comparado a sistemas que dependem de conhecimento do domínio, que pode variar de acordo com a linguagem natural em questão. De fato, a quantidade de linguagens naturais existentes aliada à característica

evolutiva destas, faz com que o custo de desenvolvimento e manutenção de software seja alto quando comparado à abordagem orientada aos dados.

As abordagens mais bem-sucedidas baseadas em dados para a análise sintática pressupõem a existência de um dataset em forma de banco de árvores (treebank ou corpus). Que consiste em pares anotados de sentença e estrutura de dependência correspondente. Este conjunto é então processado utilizando um algoritmo de treino para se obter um modelo capaz de analisar novas sentenças, caracterizando assim o aprendizado supervisionado.

2.3 Aprendizado supervisionado

A maioria dos sistemas de aprendizado de máquina atuais utilizam aprendizado supervisionado: pares de entrada e saída são processados para se obter um programa (modelo) capaz de fazer uma computação em novas entradas obtendo a saída correspondente. No caso da análise de dependência, o método pressupõem que as sentenças usadas como entrada para o aprendizado de máquina foram anotadas com suas estruturas de dependência correspondentes.

Basicamente são dois problemas diferentes que precisam ser resolvidos computacionalmente na análise de dependência supervisionada. O primeiro é o problema de aprendizado, que é a tarefa de aprender um modelo de análise (parser) a partir de uma amostra representativa de sentenças e suas respectivas estruturas de dependência. O segundo é o problema de análise (parsing), que é a tarefa de aplicar o modelo aprendido à análise de uma nova sentença. Isto é, uma sentença que não estava presente no conjunto de treino. Podemos então definir estes dois problemas da seguinte maneira:

- **Aprendizado**: Dado um conjunto D de pares (S_i, G_i) onde S_i é uma sentença associada a sua árvore de dependência correta G_i com i variando de 0 a $|D|$. O problema de aprendizado consiste em derivar um modelo M que pode ser usado para analisar novas sentenças.
- **Análise**: Dado um modelo M e uma sentença S, jamais analisada anteriormente por M, o problema consistem em obter a árvore de dependência G, que corresponde à estrutura de dependência sintática correta de S.

Ao longo deste trabalho será desenvolvida uma modelagem que trata esses dois problemas.

2.4 Representação da entrada

Para que se possa processar computacionalmente uma linguagem natural é necessário criar uma representação computacional desta linguagem. Que pode ser uma representação das unidades que compõem o vocabulário da linguagem, ou seja, uma representação das palavras.

Os primeiros sistemas de análise de dependência geralmente utilizavam uma representação atômica das palavras. Neste caso cada palavra é representada por sua identidade. Na prática isso faz com que seja alocado um vetor binário do tamanho do vocabulário para representar cada palavra. Uma representação que pode ocasionar em um processamento ineficiente em virtude do seu caráter esparso.

Figura 2.1: À esquerda um exemplo de representação atômica onde cada palavra é representada pelo seu identificador único. Ao centro as palavras são representadas por por níveis contínuos de ativação. E por fim uma representação visual onde cada palavra é representada pelo contexto na qual ela é utilizada.

Em análise de dependência, o desejado é uma representação contextual, onde uma palavra é representada levando em conta o contexto em que ela geralmente é empregada. A Figura 2.1 exemplifica intuitivamente três formas de representação de palavras.

Utilizando uma rede neural simples é possível aprender a representar palavras em vetores com baixa dimensionalidade de forma automática. Como exemplo, tome uma rede neural FFNN que utiliza palavras de um vocabulário como entrada e as incorpora como vetores de um espaço vetorial. Que então calibra os pesos através de backpropagation, produzindo representação de palavras como pesos da camada de entrada. Que geralmente é referida como *Embedding Layer*.

Na prática cada palavra w_i é mapeada em um ponto no espaço vetorial com d dimensões: $w_i \in \mathbb{R}^d$. Assim a palavra pode ser representada como um vetor de números reais, menos esparso que os vetores da representação atômica

e com possibilidade de representar informação semântica sobre a palavra e seu contexto. Tornando o modelo de aprendizado mais robusto quando comparado aos modelos que utilizam a representação atômica.

Uma das principais contribuições do trabalho de Chen e Manning 2014 foi mostrar os benefícios de se carregar a camada de entrada do classificador com a incorporação *word embedding* pré-treinada. Esta é especialmente eficaz na tarefa de análise de dependência, onde geralmente os conjuntos de treino não dispõem de milhões de exemplos anotados. Trazendo a oportunidade de lidar com este problema da escassez de exemplos através da transferência de conhecimento de um grande conjunto não anotado.

Neste contexto, a ferramenta Word2vec (Mikolov et al. 2013) se torna fundamental. Pois permite treinar em um conjunto grande e não anotado de forma computacionalmente eficiente e direcionada a extrair ligações semânticas entre palavras. Implementando diversos modelos, dentre os quais podemos destacar: *Continuous Bag of Words* (CBOW) e *Sikp-gram*. Ambos são baseados no modelo de linguagem de rede neural *Feedforward* (FFNN). O CBOW tem como objetivo prever a palavra atual usando seu contexto, enquanto o *Skip-gram* tenta prever as palavras no contexto, considerando a palavra-alvo.

2.5 Rede neural artificial

A análise de dependência vem se beneficiando da estratégia que consiste em utilizar uma modelagem de linguagem baseada em rede neural inicializado com vetores de palavras obtidos através de pré treino. Que pode ser realizado em um conjunto de dados em forma de texto plano, mais fáceis de conseguir em grande quantidade que os dados de treino para a análise de dependência. Dado que estes últimos dependem da anotação de linguistas com conhecimento em gramática de dependência.

Esta seção apresenta o conceito de uma rede neural simples, que é utilizada posteriormente na apresentação de uma modelagem do sistema de análise de dependência baseada em transição.

2.5.1 Perceptron

Perceptron é um termo muito utilizado na literatura de aprendizado de máquina para designar diversos conceitos. Pode se referir ao modelo matemático simplificado de um neurônio biológico, ou seja: a unidade fundamental de uma rede neural artificial.

A partir do neurônio artificial é possível definir um algoritmo para se aprender um classificador binário: o algoritmo Perceptron. Que pode generalizar de forma natural para diversas variantes tal como como por exemplo o perceptron multi classe ou estruturado. A Figura 2.2 apresenta a ilustração de um neurônio artificial.

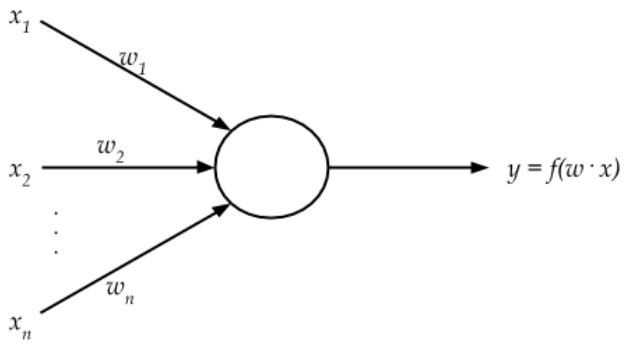


Figura 2.2: O Perceptron recebe como entrada um vetor $x = x_1, x_2, .., x_n$ e produz como saída um valor y em função do produto escalar entre x e o vetor de pesos $w = w_1, w_2, .., w_n$: $\sum_{i=1}^{n} w_i x_i$.

O valor de y é resultado da aplicação de uma função f, chamada de função de ativação, que pode ser por exemplo a função degrau. E o cálculo pode lavar em conta um valor chamado de viés (representado por b na equação abaixo) que independe da entrada.

$$y = \begin{cases} 1, & \text{se } w \cdot x + b > 0. \\ 0, & \text{caso contrário.} \end{cases}$$

Quando o valor de $w \cdot x + b$ satisfaz um determinado valor limite, que é zero no exemplo da função degrau, diz-se que o neurônio se ativa e dispara o valor 1, caso contrário diz-se que o neurônio se mantém inativo.

Esta é uma modelagem muito simples que apresenta limitações, dentre as quais a incapacidade de resolver problemas que não são linearmente separáveis. Mas que pode se tornar poderosa quando se combina um grande número de neurônios organizados em camadas.

2.5.2 Feedforward Neural Network (FFNN)

Feedforward Neural Network (FFNN) é a forma mais simples de uma rede constituída de neurônios artificiais como o da Figura 4. Nesta rede eles são organizados em camadas, em que todos os neurônios de uma camada estão conectados aos neurônios da próxima camada. A primeira camada da rede é

chamada de camada de entrada e a última camada é a camada de saída. Como mostrado na Figura 2.3.

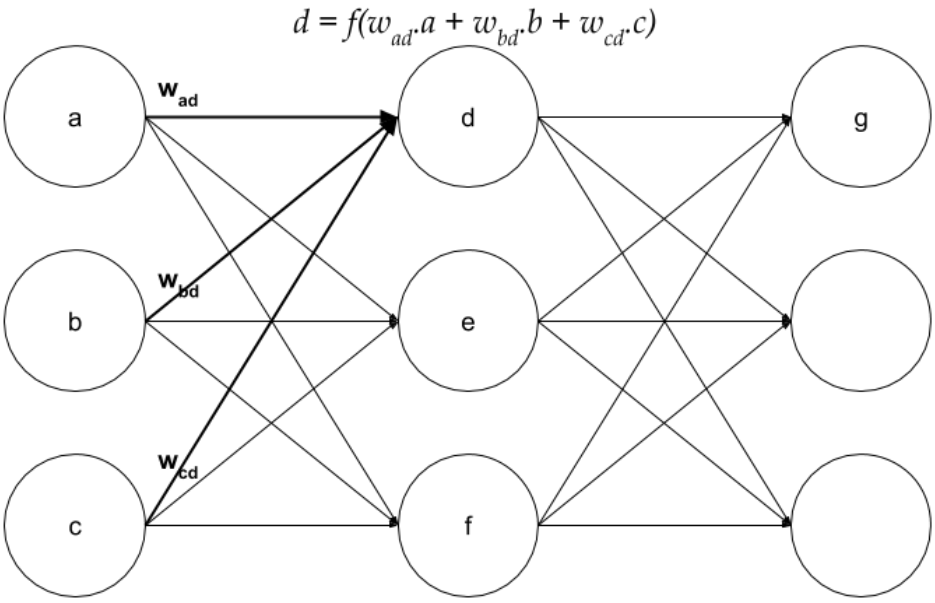


Figura 2.3: Com exceção das camadas de entrada e de saıda, todas as camadas da rede são chamadas camadas escondidas (*Hidden layers*). Os neurônios da primeira camada representam o vetor de entrada $x = a, b, c$, portanto não computam ativação.

Nesta rede, o cálculo da ativação é realizado sempre para frente, isto é, da camada de entrada para a camada de saída. Sem ciclos ou loops. Computando a saída dos neurônios das camadas em sequência. No que chamamos de *Forward Propagation*, ilustrado na Figura 2.3.

De posse de um conjunto de exemplos anotados, isto é, de pares (w, y), é possível calibrar os valores dos pesos ou parâmetros (w) através de um algoritmo de aprendizado supervisionado. O algoritmo mais popular é o algoritmo *Backpropagation*. Ele é usado para reajustar os parâmetros da rede (w) com base em uma medida de erro. A medida do erro pode ser por exemplo a diferença entre a saída obtida e a saída desejada. Usando a informação do erro, podemos percorrer a rede de trás pra frente ajustando os pesos de cada conexão para reduzir o valor do erro por uma pequena quantidade. Após repetir esse processo para um número suficientemente grande de ciclos de treinamento, a rede converge para algum estado em que o erro é pequeno. Nesse caso, dizemos que a rede aprendeu uma função.

2.6 Estruturas sintáticas

No contexto do aprendizado de máquina, dependency parsing é modelado como um problema de aprendizado estruturado. Pois a saída de um analisador sintático não é uma classe nem um valor numérico, mas sim uma estrutura. Tal estrutura descreve relações sintáticas entre as unidades que compõem uma sentença de entrada escrita em linguagem natural.

A abordagem que utiliza a estrutura baseada nos conceitos da gramática de dependência combinada com técnicas de aprendizado de máquina é a que apresenta melhor precisão na tarefa de análise sintática automática. Além disso há também a percepção de que a gramática de dependência é mais adequada do que outras estruturas equivalentes, por exemplo, a gramática de estrutura de frase, para idiomas com ordem de palavras livre ou flexível. Possibilitando assim, a análise de idiomas tipologicamente diversos utilizando a mesma modelagem.

2.6.1 Gramática de dependência

O ponto de partida do suporte teórico moderno da gramática de dependência é o trabalho de Tesnière 1959. As teorias gramaticais e formalismos que estão incluídos nesta tradição, compartilham as mesmas suposições básicas.

O pressuposto básico relativo a todas as variedades de gramática de dependência é a ideia de que a estrutura sintática consiste essencialmente em palavras ligadas por relações binárias e assimétricas chamadas relações de dependência. Uma relação de dependência se mantém entre uma palavra sintaticamente subordinada, chamada de dependente, e outra palavra da qual ela depende, chamada de cabeça sintática. A Figura 2.4 ilustra uma estrutura de dependência correspondente a seguinte sentença da língua portuguesa: *"A onça corre muito rapidamente"*.

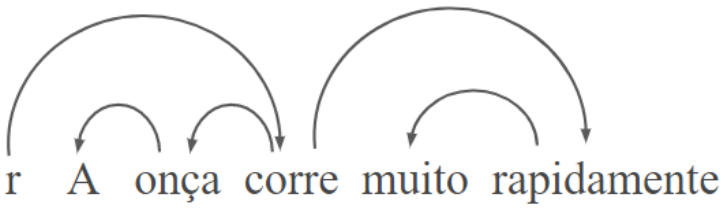


Figura 2.4: Árvore de dependência sem rótulos e com uma raiz artificial, denotada por r.

As relações de dependência são representadas por setas apontando da cabeça para o dependente. No exemplo da Figura 2.4, há uma relação entre *onça* e *corre*. Neste caso, a palavra *onça* está subordinada a *corre*, ou seja, *onça* é uma unidade linguística subordinada a *corre*, que é a cabeça sintática desta relação.

É usual a inserção de uma raiz de palavra artificial antes da primeira palavra da sentença. Como se pode observar na estrutura de dependência ilustrada na Figura 3. Em que a raiz artificial é representada pelo caractere r. Em particular, normalmente podemos supor que toda palavra real da sentença deve ter uma cabeça sintática. Assim, em vez de dizer que o verbo não tinha uma cabeça sintática, podemos dizer que é um dependente da raiz artificial da palavra. Este é um truque que simplifica as implementações computacionais.

A estrutura de dependência é útil para resolver diversos problemas NLP e pode ser modelada de forma ter baixo custo computacional. A Figura 2.5 ilustra uma representação interna de uma árvore de dependência em um sistema hipotético.

índex	0	1	2	3	4	5
S	ROOT	A	onça	corre	muito	rapidamente
head		2	3	0	5	3

Figura 2.5: Estrutura interna da sentença S ="A onça corre muito rapidamente". As palavras da frase têm um índice que é referenciado no vetor principal. O sistema representa uma árvore de dependência referindo-se apenas aos índices, que podem ser representados por inteiros de 8 bits quando as sentenças são formadas por um número limitado de palavras.

A relação de dependência pode estar decorada com um rótulo que classifica o tipo da relação, como no exemplo da Figura 1.2. Neste exemplo, o substantivo clima é um dependente do verbo ser (em sua forma é) com tipo de dependência sujeito (designado pelo rótulo nsubj).

Formalmente, $S = w_0 w_1 ... w_n$ denota uma sentença escrita em linguagem natural formada pelas unidades sintáticas w_x com $x \in [0, n]$. Naturalmente, para as língua portuguesa e inglesa, tais unidades sintáticas são palavras ou símbolos de pontuação.

2.6.2 Grafos e árvores

Como visto anteriormente, a estrutura sintática baseada em dependência de uma determinada sentença é modelada como um grafo enraizado. Os nós

representam os elementos lexicais da sentença e as arestas representam as relações de dependência entre esses elementos. As relações de dependência são direcionadas das cabeças aos dependentes. Aqui usamos o termo token para referenciar as unidades sintáticas que compõem a sentença.

Seja $S = w_0 w_1 ... w_n$ uma sentença de entrada, em que $w_i, i \in [1, n]$ é o i-ésimo token em S e w_0 é um token artificial que representa a raiz. A estrutura sintática associada a S é um grafo direcionado $G = (V, A)$ onde existe um nó i para cada token w_i, então $V \subseteq \{0, 1, ...n\}$; e várias arestas $(i, j) \in A \subseteq V \times V$ que representam as relações de dependência. Quando $(i, j) \in A$ isso significa que existe uma relação de dependência entre o token dependente w_j e sua cabeça sintática w_i. A Figura 1.2 exemplifica essa definição. Em particular, uma aresta (i, r, j) pode ser rotulada por r com a relação de dependência sintática entre o token w_i e w_j. Quando a estrutura não tem nenhuma preocupação com esses rótulos, ela é chamada de estrutura de dependência não rotulada.

Para fornecer uma análise sintática completa de uma frase, o grafo deve ser conexo. Esta restrição específica que cada nó está relacionado a pelo menos um outro nó. A maioria das versões de gramáticas de dependência, assim como UD, pressupõe que cada nó tenha no máximo uma cabeça sintática, isso é chamado de restrição de cabeça única. Outra suposição é que o grafo não deve conter ciclos, essa é a restrição de aciclicidade. Trata-se portanto de um grafo direcionado sem ciclos (DAG). Isso, juntamente com o fato de ser conexo, implica que o grafo deve ser uma árvore enraizada, com uma raiz única que não é dependente de nenhum outro nó. Portanto, essa estrutura restrita é chamada de árvore de dependência.

Outra preocupação em relação à representação formal de dependências é a relação entre estrutura de dependência e ordem de tokens. O exemplo mais conhecido é a restrição da projeção. Um grafo de dependência satisfaz a restrição de projetividade em relação a uma ordem linear particular dos nós se, para cada aresta (h, d) e nó w, w ocorre entre h e d em uma ordem linear somente se w é dominado por h (onde domina é o fechamento reflexivo e transitivo da relação arco). A estrutura representada na Figura 1.2 representa uma árvore de dependência projetiva enquanto que a Figura 2.6 apresenta um exemplo de estrutura não projetiva. É notório que a restrição da projetividade é demasiadamente rígida para a descrição de linguagens com ordem de palavras livres, como a língua portuguesa.

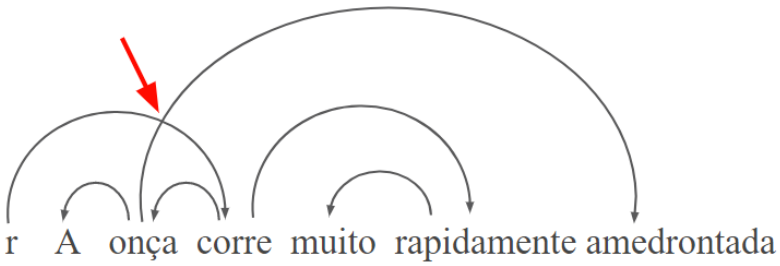


Figura 2.6: Um exemplo de árvore de dependência não projetiva. Uma árvore é projetiva se todos os arcos são projetivos. Um arco da cabeça para o dependente é projetivo se houver um caminho da cabeça para todos os tokens dentro do arco. Este não é o caso do arco entre a cabeça "onça"e o dependente "amedrontada"neste exemplo. A seta vermelha indica arcos que visualmente se cruzam, uma característica de árvores de dependência não projetivas.

2.7 Universal dependencies (UD)

Relacionar a precisão de um parser a um conjunto de propriedades linguísticas pode revelar propriedades interessantes. Contudo, dadas as importantes diferenças tipológicas que existem entre as línguas, bem como a diversidade de esquemas de anotação usados em diferentes bancos de árvores, comparar essas categorias entre idiomas diferentes não é uma tarefa simples.

Universal Dependencies (De Marneffe et al. 2014; Nivre 2015) desenvolve uma representação de dependência sintática e recursos que podem ser utilizados com fidelidade lingüística e usabilidade humana em diversas linguagens naturais. Isto é, propõe um esquema de anotação de banco de árvores (“treebank”, corpus) projetado para ser consistente dentre os vários idiomas existentes. Por exemplo, para tipos de relação de dependência, podemos considerar uma categoria raiz geral (para rótulos usados em arcos a partir da raiz artificial, incluindo um rótulo genérico ou o rótulo atribuído a predicados de cláusulas principais, que são normalmente verbos), uma categoria de assunto e um objeto categoria (incluindo objetos diretos e indiretos), todos exemplos de categorias amplas que são consistentes entre as línguas.

Construído sobre formatos herdados das dependências de Stanford, CoNLL, Google UPOS e outros formatos, a iniciativa Universal Dependencies (UD) representa um esforço para tornar o processamento de linguagem natural multilingue mais coeso. É produzido e mantido através de uma comunidade de código livre[1]. Disponibiliza em torno de 100 datasets em mais de 60 idiomas, e este número continua crescendo.

[1] http://universaldependencies.org/

Analisar as Dependências Universais é uma das tarefas compartilhadas das conferências CoNLL de 2017 e 2018. Os conceitos utilizados pelos principais concorrentes da tarefa compartilhada foram levados em consideração ao escolher as especificações a serem seguidas. Além disso, este trabalho usa os mesmos conjuntos de dados utilizados na conferência para treinar e testar nosso sistema, bem como o mesmo script de validação.

2.8 O formato CoNLL-U

Desde 2006, quando pela primeira vez o foco da tarefa compartilhada da CoNLL foi na análise de dependência multilíngue, o formato CoNLL foi sendo ampliado e revisado. O formato de representação adotado pelo UD hoje é uma versão revisada deste formato original. Ele é um arquivo texto simples composto de três tipos de linhas:

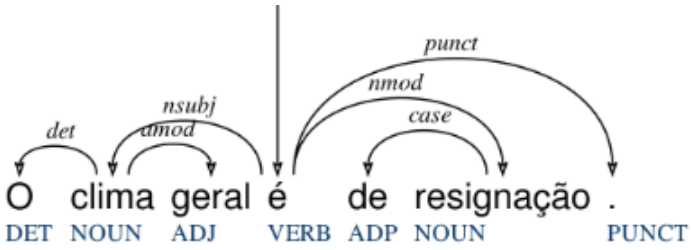
- Linhas em branco, que separam as sentenças.
- Linhas de comentário, que começam com o caractere #.
- Linhas de palavras, com dez campos separados por tabulação simples:

    1. ID: Índice do elemento sintático, que pode ser uma palavra ou símbolo de pontuação, no caso de uma sentença em língua portuguesa. É representado por um número inteiro começando em 1 para cada nova sentença; pode ser um intervalo para tokens de várias palavras; pode ser um número decimal para nós vazios.
    2. FORM: Forma do elemento sintático ou símbolo de pontuação.
    3. LEMMA: Lema ou radical da palavra.
    4. UPOSTAG: Rótulo POS (part-of-speech) Universal.
    5. XPOSTAG: Rótulo POS específico do idioma; sublinhado ("_") se não estiver disponível.
    6. FEATS: Lista de características morfológicas do inventário de recursos universal ou de uma extensão específica do idioma; sublinhado ("_") se não estiver disponível.
    7. HEAD: Cabeçalho da palavra atual, que é um valor de ID ou zero (0) para designar a raiz artificial.
    8. DEPREL: Relação de dependência universal com respeito ao HEAD ("root" se, e somente se, HEAD = 0) ou um subtipo específico de idioma definido.

9. DEPS: Grafo de dependência aprimorado na forma de uma lista de pares de (HEAD-DPREL).
10. MISC: Qualquer outra anotação.

Os campos ID e HEAD definem uma relação de dependência, enquanto DEPREL define seu rótulo. Estes são os campos preenchidos com a tarefa de análise de dependência. Que depende da Segmentação, onde são preenchidos os campos 1, 2 e 10. E da tarefa de Tagging, em que são preenchidos os campos 3, 4, 5 e 6. Podendo utilizar portanto como entrada informação de todos esses campos.

A Figura 2.7 mostra uma árvore de dependência e sua representação correspondente no formato CoNLL-U.



ID	FORM	LEMMA	UPOSTAG	XPOSTAG	FEATS	HEAD	DEPREL	DEPS	MISC
1	O	_	DET	DET	_	2	det	_	_
2	clima	_	NOUN	NOUN	_	4	nsubj	_	_
3	geral	_	ADJ	ADJ	_	2	amod	_	_
4	é	_	VERB	VERB	_	0	root	_	_
5	de	_	ADP	ADP	_	6	case	_	_
6	resignação	_	NOUN	NOUN	_	4	nmod	_	
7	.	_	PUNCT	.	_	4	punct	_	_

Figura 2.7: Árvore de dependência com rótulos e tag POS (UPOSTAG) e sua representação no formato CoNLL-U. Exemplo retirado do conjunto de treino da língua portuguesa (UD 2.0 corpus pt-br).

2.9 Métricas de avaliação

A metodologia padrão para avaliar parsers, é aplicá-lo a um conjunto de teste extraído de um banco de árvores e comparar a árvore de dependência gerada pelo analisador (*parser*) com a árvore de dependência anotada no banco (*gold tree*). Sendo assim, a qualidade de um parser é dada pela similaridade

entre as árvores de dependência que ele produz e as árvores de dependência anotadas correspondentes.

Intuitivamente podemos medir o percentual de sentenças que apresentam árvores produzidas pelo parser idênticas as árvores anotadas. Entretanto, dado que uma sentença é representada por uma estrutura, uma alternativa mais comum é obter uma medida baseada não na árvore completa, mas na quantidade de nós que a compõem.

Nesse sentido as métricas mais utilizadas atualmente são os chamados Attachment Scores. Onde dois tipos prevalecem:

– Labeled Attachment Score (LAS): percentual de nós atribuídos com pares corretos de cabeça sintática e rótulo de dependência.

– Unlabeled Attachment Score (UAS): análogo ao LAS, mas leva em conta apenas as cabeças sintáticas, desprezando os rótulos das relações.

Embora ambas as métricas sejam muito utilizadas em análise de dependência, na edição de 2018 da tarefa compartilhada na CoNLL apenas a métrica LAS foi considerada, juntamente com outras duas métricas (MLAS e BLEX).[2]

Este trabalho considera as duas métricas: UAS e LAS.

2.10 Enfoque

As abordagens orientadas aos dados diferem no tipo de modelo de análise (parser) adotado, nos algoritmos usados para aprender o modelo a partir dos dados e nos algoritmos usados para analisar novas sentenças com o modelo. Destacando-se duas classes de métodos orientados aos dados, onde a maior parte da pesquisa em análise de dependência tem o seu foco. Os quais chamamos de baseados em transição (*Transition-based Dependency Parsing*) e baseados em grafos (*Graph-based dependency parsing*).

2.10.1 O método baseado em grafo

Nesta seção encontram-se alguns comentários sobre a estratégia utilizada na abordagem baseada em grafo. Uma introdução mais completa sobre esta modelagem pode ser encontrada em Barroso 2016.

Analisadores baseados em grafos tratam a análise de dependência como um problema de aprendizado estruturado baseado em pesquisa, no qual o objetivo é aprender uma função de pontuação sobre árvores de dependência.

[2] http://universaldependencies.org/conll18/evaluation.html

De modo que a árvore de dependência correta obtenha uma pontuação superior a todas as árvores possíveis para uma dada sentença de entrada.

A estratégia se desenvolve através da construção de um grafo completo correspondente a todas as possíveis relações sintáticas de uma dada sentença. Em seguida é utilizado um classificador para atribuir pesos nas arestas do grafo. E finalmente a partir deste grafo encontra-se a arborescência geradora máxima. Sendo esta arborescência eleita como a saída ótima do parser.

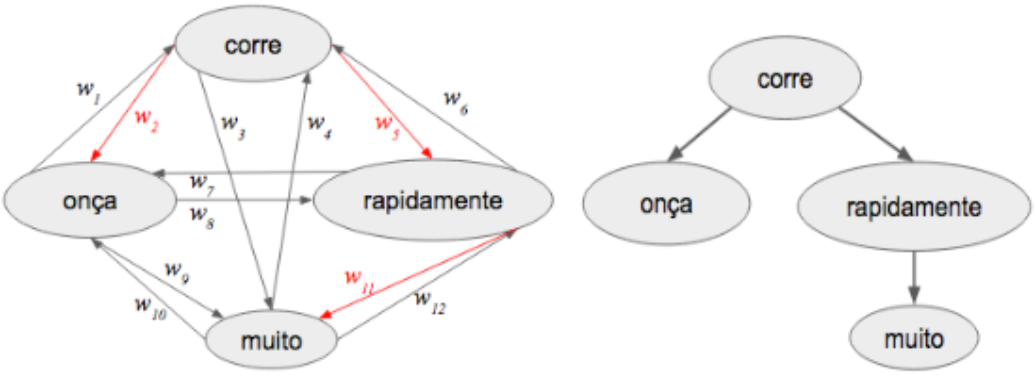


Figura 2.8: À esquerda um grafo direcionado completo com pesos nas arestas, e à direita uma arborescência geradora deste grafo.

A figura 2.8 apresenta uma ilustração intuitiva de estruturas manipuladas em um sistema orientado a grafo efetuando a análise sintática da sentença: *"Onça corre muito rapidamente"*. Primeiro o sistema constrói o grafo completo com relações entre as unidades sintáticas para então encontrar a árvore de dependência.

Sistemas deste tipo podem utilizar uma busca exaustiva no espaço de árvores possíveis para encontrar a solução ótima. Sendo geralmente mais precisas do que sistemas baseados em transição, mas em contrapartida apresentam menor eficiência. Existem algoritmos que visam melhorar a eficiência limitando a complexidade da busca em tempo polinomial com relação ao comprimento da sentença de entrada.

2.10.2 O método baseado em transição

A análise de dependência baseada em transição formaliza o problema de análise como uma série de decisões, ou transições, que leem palavras sequencialmente de um buffer e as combinam de forma incremental em estruturas sintáticas para chegar na estrutura pretendida: a árvore de dependência.

A modelagem se desenvolve através da definição de uma máquina abstrata semelhante a um autômato de pilha, em que as transições correspondem

a etapas na derivação de uma árvore de dependência. Assim, o problema de aprendizado consiste em induzir um modelo capaz de prever a próxima transição, e o problema de análise é construir a sequência de transições ótima para uma dada sentença de entrada. Esta estratégia é o foco deste trabalho, e será apresentada em mais detalhes a seguir.

3
O método baseado em transição

Analisadores baseados em transição tratam o problema da análise de dependência como uma sequência de ações que produzem uma árvore correspondente a análise de um texto. Um classificador é treinado para pontuar as ações possíveis em cada estágio do processo e guiar o processo de análise.

Os analisadores básicos baseados em transição funcionam de maneira gulosa, realizando uma série de decisões ótimas localmente e possuem velocidades de análise muito rápidas quando comparada a outros métodos. Em contraste, analisadores baseados em transição mais avançados introduzem alguma pesquisa no processo usando por exemplo uma busca em feixe (Zhang e Clark 2008) e programação dinâmica (Huang e Sagae 2010).

Este capítulo visa explicar o funcionamento de uma modelagem simples baseada em transição. Iniciando com a apresentação de uma estratégia baseando-se no trabalho de Kubler et al. 2009. Mostra como um oráculo é usado no momento do treinamento para mapear as configurações do analisador em transições ideais com respeito a uma árvore de dependência correta. E como um classificador é treinado para emular as previsões do oráculo. Constituindo assim o que chamamos de análise de dependência baseada em transição.

Podemos então elencar alguns conceitos importantes que serão abordados no decorrer deste capítulo:

- Sistema de transição: definições e exemplos de alguns dos sistemas de transição mais utilizados em análise de dependência.
- Atributos: em sistemas baseados em transição, atributos são extraídos a partir de uma configuração do sistema de transição.
- Classificador: utilizado para decidir qual transição aplicar em uma determinado momento da análise. Ou seja, como é feito o mapeamento da representação dos atributos em transições.
- Oráculo: componente responsável por mapear a estrutura sintática à uma sequência de transições corretas com respeito a tal estrutura.
- Dataset: obtenção de datasets de código livre através da iniciativa UD. Como preparar para o treino, isto é, como associar estruturas sintáticas à sequências de transições utilizando o Oráculo.

Ao final serão apresentadas algumas alternativas ao método inicialmente proposto no que diz respeito ao oráculo, sistema de transição e estratégia de treinamento.

3.1 Sistemas de transição

A modelagem baseada em transição parametriza um modelo de aprendizado sobre transições de uma máquina abstrata. Esta máquina abstrata, que é similar a um autômato de pilha, recebe o nome de sistema de transição. Os estados da máquina são chamados de configurações, e suas operações chamadas de transições. O problema de aprendizado consiste em aprender a prever a próxima transição dada a entrada e o histórico de análise, e onde se prediz novas árvores usando um algoritmo de análise, que pode ser guloso e determinístico.

Um dos exemplos mais simples é um autômato de estado finito, que consiste em um conjunto finito de estados atômicos e transições definidos em estados e símbolos de entrada. E que aceita uma string de entrada se houver uma sequência de transições válidas de um estado inicial designado para um dos vários estados terminais. Em contraste, os sistemas de transição usados para análise de dependência têm configurações complexas com estrutura interna, em vez de estados atômicos. E transições que correspondem a etapas na derivação de uma árvore de dependência. A ideia é que uma sequência de transições válidas, iniciando na configuração inicial de uma determinada sentença e terminando em uma das várias configurações terminais, define uma árvore de dependência válida para a sentença de entrada.

Um sistema de transição possui uma memória (buffer), com a sentença de entrada segmentada e utiliza uma pilha como memória auxiliar enquanto deriva um conjunto de arcos de dependência. Conforme ilustrado anteriormente na figura 3.1.

Seja a sentença $S = w_0 w_1 ... w_n \in \{S\}$ o conjunto de palavras que constituem S, podemos então definir uma configuração como sendo a tripla $c = (\sigma, \beta, A)$, onde:

1. σ é uma pilha de palavras $w_i \in \{S\}$
2. β é um buffer de palavras $w_i \in \{S\}$
3. A é um conjunto de dependência $(w_i, w_j) \in \{S\} \times \{S\}$

A ideia é que uma configuração represente uma análise parcial da sentença de entrada, onde as palavras na pilha σ são palavras parcialmente processadas, as palavras no buffer β são as palavras de entrada restantes que

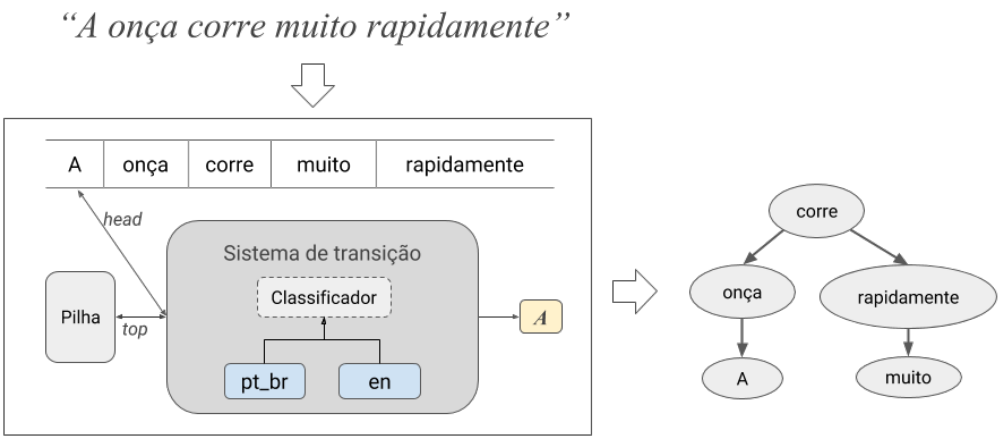


Figura 3.1: O sistema recebe como entrada uma sentença e produz como saída a árvore de dependência correspondente. Em um dado momento do processamento, o sistema apresenta configuração c_i. Que se refere ao estado de suas estruturas internas: o buffer, a pilha e o conjunto de arcos de dependência derivados A.

ainda não foram processadas, e o conjunto de arcos A representa uma árvore de dependência parcialmente construída. Por exemplo se a sentença de entrada é *"A onça corre muito rapidamente."*. Então a seguinte configuração é válida, onde a pilha contém as palavras raiz e *"rapidamente"* (com a último no topo), o buffer contém todas as palavras restantes, exceto *"muito"*, e o conjunto de arco contém um único arco conectando *"rapidamente"* ao dependente *"muito"*:

([root | rapidamente | σ] , [A | onça | corre | . | β], {(rapidamente, muito)}A)

Observe que representamos a pilha e o buffer como listas simples, com elementos entre colchetes. Com a seguinte notação: $\sigma|x$ para denotar uma pilha com o elemento x no topo e o resto σ. E $x|\beta$ para denotar um buffer com uma cabeça x seguida pelos elementos em β.

O sistema inicializa o buffer e a pilha para em seguida começar a aplicar transições. Adicionando incrementalmente arcos ao conjunto A após cada ação de transição, até que um estado terminal seja atingido.

Existem diversos sistemas de transição e eles diferem entre si pela forma com que eles definem suas configurações iniciais e terminais e pelo conjunto de transições suportadas. A seguir vamos nos concentrar em um sistema de transição simples baseado em pilha que chamaremos de Shift-reduce. Por hora nos concentramos na definição das transições e funcionamento do sistema. Utilizado para exemplificar a abordagem mais amplamente usada na análise de dependência baseada em transição. O componente Classificador será abordado

posteriormente.

No final do capítulo são apresentados alguns sistemas alternativos. Incluindo o sistema que utilizamos nos experimentos.

3.1.1 O sistema Shift-reduce

Chamaremos de *Shift-Reduce* o sistema de transição em que, dado uma sentença $S = w_0, ..., w_n$, temos:

- a configuração inicial $c_0(S)$ é dada por $([w_0]\sigma, [w_1, ..., w_n]\beta, \emptyset)$,
- a configuração terminal (ou final) é qualquer configuração da forma $(\sigma, \emptyset, A)$ para quaisquer σ e A. Ou seja, qualquer configuração em que o buffer se encontra vazio.

Assim, inicializamos o sistema para uma configuração com $w_0 = \text{ROOT}$ raiz artificial na pilha, todas as palavras restantes no buffer e um conjunto de arco A vazio; e terminamos em qualquer configuração que tenha um buffer vazio (independentemente do estado da pilha e do conjunto de arcos).

Tendo definido o conjunto de configurações, incluindo uma configuração inicial única e um conjunto de configurações de terminais para qualquer sentença, agora definimos as transições entre as configurações. Formalmente, uma transição é uma função parcial de configurações para configurações, ou seja, uma transição mapeia uma determinada configuração para uma nova configuração, mas pode ser indefinida para determinadas configurações. Conceitualmente, uma transição corresponde a uma ação de análise básica que adiciona um arco de dependência na árvore ou modifica a pilha ou o buffer. As transições necessárias para a análise de dependência neste sistema são definidas abaixo:

- $SHIFT[(\sigma, [b|\beta], A)] = ([\sigma|b], \beta, A)$
  - remove o primeiro elemento b do buffer e empilha ele em σ. Tem como única condição prévia que o buffer não esteja vazio.
- $RIGHT[([\sigma|s], [b|\beta], A)] = (\sigma, [s|\beta], A \cup (s, b))$
  - adiciona um arco de dependência (s, b) ao conjunto de arcos A, onde s é o elemento no topo da pilha e b é o primeiro elemento no buffer. Além disso, eles desempilham s e substituem b por s na cabeça do buffer. Apesar de parecer contra-intuitivo, dado que o buffer deve conter palavras que ainda não foram processadas, isso é necessário para permitir que b se conecte a uma cabeça à sua esquerda. A única

condição prévia é que tanto a pilha quanto o buffer não estejam vazios.

– $LEFT[([\sigma|s],[b|\beta],A)] = (\sigma,[b|\beta],A \cup (b,s))$

  – adiciona um arco de dependência (b,s) ao conjunto de arcos A, onde s é o elemento do topo da pilha e b é o primeiro elemento no buffer. Além disso, s é desempilhado. Eles têm como pré-condição que tanto a pilha quanto o buffer não estão vazios e que $s \neq$root.

Usamos o símbolo T para nos referirmos ao conjunto de transições permitidas em um determinado sistema de transição. Conforme observado acima, as transições correspondem a ações de análise elementares. Para definir as análises completas, introduzimos a noção de uma sequência de transição.

Uma sequência de transição para uma sentença $S = w_0 w_1 ... w_n$ é uma sequência de configurações $C_0, m = (c_0, c_1, ..., c_m)$ tal que:

– c_0 é a configuração inicial $c_0(S)$ para S
– c_m é uma configuração terminal
– Para cada i tal que $1 \leq i \leq m$, existe uma transição $t \in T$ tal que $c_i = t(c_{i-1})$.

Uma sequência de transição começa na configuração inicial de uma determinada sentença e atinge uma configuração terminal aplicando transições válidas de uma configuração para a seguinte. A árvore de dependência derivada através desta sequência de transição é a árvore de dependência definida na configuração terminal, isto é, a árvore $G_{cm} = (VS, A_{cm})$, onde A_{cm} é o conjunto de arcos definido na configuração terminal c_m. A figura 3.2 ilustra a execução de análise da sentença com este sistema, mostrando também uma representação gráfica da construção da árvore de dependência.

O sistema de transição definido para análise de dependência nesta seção leva a derivações que correspondem à análise básica de shift-reduce para gramáticas livres de contexto. As transições LEFT e RIGHT correspondem a ações de redução, substituindo uma estrutura dependente da cabeça por sua cabeça, enquanto a transição SHIFT equivale a consumir um elemento do buffer, empilhando-o sem que seja derivado um novo arco de dependência.

Cada sequencia de transição neste sistema define um grafo enraizado que respeita às propriedades das relações de dependência, mas não necessariamente é um grafo conexo. Isto significa que nem toda sequência de transição define uma árvore de dependência, como definido no capítulo 2, mas sim uma floresta de dependência. Para pegar um exemplo trivial, uma sequencia de transição

Ação	Pilha	Buffer	Derivação
	[ROOT]	[A | onça | corre | muito | rapidamente]	[]
SHIFT	[ROOT | A]	[onça | corre | muito | rapidamente]	[]
LEFT	[ROOT]	[onça | corre | muito | rapidamente]	[(onça, A)]
SHIFT	[ROOT | onça]	[corre | muito | rapidamente]	[(onça, A)]
LEFT	[ROOT]	[corre | muito | rapidamente]	[(onça, A), (corre, onça)]
SHIFT	[ROOT | corre]	[muito | rapidamente]	[(onça, A), (corre, onça)]
SHIFT	[ROOT | corre | muito]	[rapidamente]	[(onça, A), (corre, onça)]
LEFT	[ROOT | corre]	[rapidamente]	[(onça, A), (corre, onça), (rapidamente, muito)]
RIGHT	[ROOT]	[corre]	[(onça, A), (corre, onça), (rapidamente, muito), (corre, rapidamente)]
SHIFT	[ROOT | corre]	[]	[(onça, A), (corre, onça), (rapidamente, muito), (corre, rapidamente)]

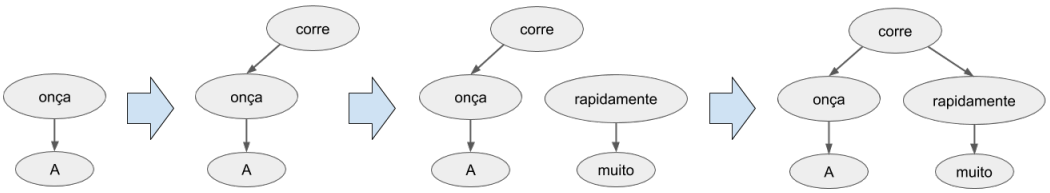


Figura 3.2: Construção da árvore de dependência passo a passo com o sistema shift-reduce. A primeira linha da tabela mostra a configuração inicial com a pilha contendo a raiz artificial e o buffer preenchido com todos os elementos da sentença de entrada. Ações de transição são aplicadas até que o buffer fique vazio. Configurando assim um estado terminal, que é a condição de parada do sistema.

para uma sentença S consistindo apenas de transições de SHIFT define o grafo $G = (V_S, \emptyset)$, que não é conexo, mas que satisfaz todas as outras propriedades. No entanto, uma vez que qualquer sequência de transição define um grafo acíclico de dependência G, é trivial converter G em uma árvore de dependência G' adicionando arcos da forma (raiz, w_i) para cada w_i que é uma raiz em G. Ou seja, retorna uma árvore de dependência adicionando um arco de dependência da raiz artificial à raiz de cada árvore da floresta.

Outra propriedade importante do sistema é que toda sequencia de transição define uma floresta de dependência projetiva, que é vantajosa do ponto de vista da eficiência, mas excessivamente restritiva do ponto de vista da representatividade. Na seção 3.8.3 apresentamos a definição de um sistema de transição alternativo capaz de vencer tal restrição de projetividade.

Dado que cada sequencia de transição define uma floresta de dependência projetiva, que pode ser transformada em uma árvore de dependência, dizemos que o sistema é sólido em relação ao conjunto de árvores de dependência projetiva. Uma questão natural é se o sistema também está completo com relação a essa classe de árvores de dependência, ou seja, se toda árvore de dependência projetiva pode ser definida através de alguma sequência de transição. A resposta a esta pergunta é afirmativa, como demonstrado em Nivre 2008.

Nosso modelo de análise de dependência pode então ser definido como $M = (T, \lambda, h)$, onde T é o conjunto de definições de transições, λ é o conjunto de parâmetros que será calibrado a partir dos dados e h é um algoritmo fixo de análise.

3.2 Análise de dependência utilizando transições

O sistema de transição definido na seção anterior é não-determinístico no sentido de que geralmente há mais de uma transição válida para qualquer configuração não-terminal. Assim, a fim de realizar a análise determinística, precisamos de um mecanismo para responder:

Dada qualquer configuração não-terminal c, qual é a transição correta a ser aplicada em c?

Suponhamos, por enquanto, que nos seja dado um oráculo, isto é, uma função de configurações para transições tal que $o(c) = t$ se e somente se t é a transição correta dado c. De posse deste tal oráculo, a análise determinística pode ser obtida pelo algoritmo 1.

Algorithm 1: Análise determinística com oráculo

input : S, T, o
output: G_c
$c \leftarrow c_0(S)$;
while *c is not terminal* **do**
  $t \leftarrow o(c)$;
  $c \leftarrow t(c)$;
end

Começamos na configuração inicial $c_0(S)$ e, desde que não tenhamos atingido uma configuração terminal, usamos o oráculo para encontrar a transição ótima $t = o(c)$ e aplicá-lo à nossa configuração atual para alcançar a próxima configuração $t(c)$.

Quando chegamos a uma configuração de terminal, simplesmente retornamos a árvore de dependência definida pelo nosso conjunto de arco atual. Note que enquanto o problema de encontrar a transição ótima $t = o(c)$ se trata de um problema difícil, que precisa ser tratado com aprendizado de máquina, computar a próxima transição é uma operação trivial.

É fácil mostrar que, desde que haja pelo menos uma transição válida para cada configuração não-terminal, tal analisador irá construir exatamente uma seqüência de transição $C_{0,m}$ para uma sentença S e retornará a árvore de dependência definida pela configuração terminal cm, isto é, $G_{cm} = (V_S, A_{cm})$. Para ver que há sempre pelo menos uma transição válida para com uma

configuração não-terminal, só temos que observar que tal configuração deve ter um buffer não vazio, o que significa que pelo menos $SHIFT$ é uma transição válida.

A complexidade de tempo do algoritmo de análise determinístico baseado em transição é $O(n)$, onde n é o número de palavras na sentença de entrada S, desde que as funções oráculo e transição possam ser calculadas em tempo constante. Isto é válido porque o pior tempo de execução é limitado pelo número máximo de transições em uma seqüência de transição $C_{0,m}$ para uma sentença $S = w_0w_1...w_n$. Isso pode ser verificado analisando as transições, que, ou diminuem o comprimento do buffer ou deixam-no inalterado. Como uma transição $SHIFT$ diminui o comprimento do buffer em 1, e nenhuma outra transição aumenta o tamanho do buffer, e ainda, qualquer configuração com um buffer vazio é terminal, o número de transições $SHIFT$ em $C_{0,m}$ é limitado por n. Além disso, como $LEFT$ e $RIGHT$ diminuem a altura da pilha em 1, somente $SHIFT$ aumenta a altura da pilha em 1, e a altura inicial da pilha é 1, o número combinado de instâncias de $LEFT$ e $RIGHT$ em $C_{0,m}$ também é limitado por n. Portanto, a pior complexidade do tempo é $O(n)$.

Até agora, vimos como a análise baseada em transição pode ser executada em tempo linear se restrita a árvores de dependência projetiva e desde que tenhamos um oráculo em tempo constante que prevê a transição correta a partir de qualquer configuração não-terminal. Para construir sistemas de análise prática, precisamos encontrar algum mecanismo que possamos usar para aproximar o oráculo o suficiente para tornar a análise viável. Há muitas formas concebíveis de aproximar oráculos, incluindo o uso de gramáticas formais e heurísticas diversas. No entanto, a estratégia mais bem sucedida até agora em análise de dependência baseada em transição tem sido adotar uma abordagem baseada em dados, aproximando oráculos de classificadores treinados em dados de banco de árvores.

Na prática, um oráculo capaz de prever transições corretas é possível de se obter, como veremos a seguir, mas seu escopo é restrito ao conjunto de sentenças do banco de árvores. Pois ele funciona baseado em informação extraída das anotações do banco de árvores. Para que possamos analisar novas sentenças, treinamos um classificador em um conjunto de decisões gerado com o oráculo. Ou seja, aprendemos um classificador capaz de se comportar como o oráculo em sentenças inéditas com certo grau de qualidade. Isso leva à noção de análise baseada em classificador, que é um componente essencial da análise de dependência baseada em transição.

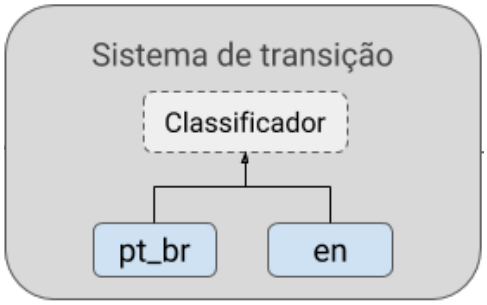


Figura 3.3: Sistema de transição com classificador.

3.3 Classificador

Retornando a caracterização geral do modelo como $M = (T, \lambda, h)$, onde T é o conjunto de transições, λ é um conjunto de parâmetros do modelo e h é um algoritmo de análise fixo. Nas duas seções anteriores, mostramos como podemos definir o algoritmo de análise h de forma gulosa em um sistema de transição (embora outras estratégias de busca sejam possíveis).

O sistema de transição determina o conjunto de restrições, mas também define os parâmetros do modelo λ que precisam ser aprendidos dos dados, pois precisamos ser capazes de prever a transição oráculo $o(c)$ para cada configuração possível c (para qualquer sentença de entrada S). Para isto, precisamos obter o componente classificador, ilustrado na figura 3.3.

Usamos a notação $\lambda_c \in \lambda$ para denotar a transição prevista para c de acordo com os parâmetros do modelo λ, e podemos pensar em λ como uma grande tabela contendo a transição prevista λ_c para cada configuração possível c. Na prática, λ é normalmente uma representação compacta de uma função para calcular λ_c dado c, mas os detalhes dessa representação não precisam nos preocupar agora. Dado um modelo aprendido, podemos realizar análise determinística baseada na transição usando o algoritmo 2, onde simplesmente substituímos a função oráculo pelos parâmetros aprendidos λ (e o valor da função $o(c)$ pelo valor do parâmetro específico λ_c).

No entanto, para tornar o problema de aprendizagem tratável por técnicas padrão de aprendizado de máquina, precisamos introduzir uma abstração sobre o conjunto infinito de configurações possíveis. Isto é feito através de uma função $\varphi(x) : X \to Y$. No nosso caso, o domínio X é o conjunto C de configurações possíveis (para qualquer sentença S) e o intervalo Y é um produto de m conjuntos de valores de atributos, o que significa que a função de atributo $\varphi(c) : C \to Y$ mapeia todas as configurações para um vetor de atri-

Algorithm 2: Algoritmo determinístico de análise de dependência com classificador.

```
   input  : S, T, λ
   output: G_c
1  c ← c_0(S);
2  while c is not terminal do
3  |  t ← λ(c);
4  |  c ← t(c);
5  end
```

butos m-dimensional. Dada esta representação, queremos então aprender um classificador $g : Y \rightarrow T$, onde T é o conjunto de transições possíveis, tal que $g(\varphi(c)) = o(c)$ para qualquer configuração c. Em outras palavras, dado um conjunto de treinamento para análise de dependência (treebank), queremos aprender um classificador que prevê a transição do oráculo $o(c)$ para qualquer configuração c, dado como entrada a representação de atributos $\varphi(c)$. Isso dá origem a três questões básicas:

- Como podemos representar as configurações utilizando vetores de atributos?
- Dado que o conjunto de dados não possui transições, como derivamos os dados de treinamento para conter transições?
- Como treinamos classificadores?

Ao longo deste capítulo serão abordadas cada uma dessas questões. A representações de atributos, derivação dos dados de treinamento e funcionamento dos classificadores.

3.4 Atributos

Uma representação de atributos $\varphi(c)$ de uma configuração c é um vetor multi-dimensional de atributos simples $\varphi(c)$ (para $1 \leq i \leq m$). No caso geral, esses atributos simples podem ser definidos por atributos arbitrários de uma configuração, podendo ser categórico ou numérico. Por exemplo, "a tag POS no topo da pilha" é um atributo categórico, com valores obtidos de um conjunto de tags de parte da fala em particular (por exemplo, NN para substantivo, VB para verbo, etc.). Por outro lado, "o número de dependentes anteriormente anexados à palavra no topo da pilha" é um recurso numérico, com valores retirados do conjunto $\{0, 1, 2, \dots\}$. A escolha de representações de atributos é, em parte, dependente da escolha do algoritmo de aprendizado, uma vez que alguns algoritmos impõem restrições especiais na forma que

os valores de recursos podem assumir, por exemplo, que todos os recursos devem ser numéricos. No entanto, no interesse da generalidade, vamos ignorar essa complicação por enquanto e assumir que os recursos podem ser de qualquer tipo. Isso não é problemático, pois é sempre possível converter atributos categóricos em atributos numéricos e simplificar muito a discussão de representações de atributos para análise baseada em transição.

Os atributos mais importantes na análise baseada em transição são definidos por atributos de palavras ou nós de árvore, identificados por sua posição na configuração. Geralmente, é conveniente pensar nesses atributos como definidos por duas funções mais simples, uma função de endereço identificando uma palavra específica em uma configuração (por exemplo, a palavra no topo da pilha) e uma função de atributo que seleciona um atributo específico dessa palavra (por exemplo , sua tag POS). Chamamos esses atributos de atributos de configuração de palavras e os definimos da seguinte forma.

Dada uma sentença de entrada $S = w_1, ..., w_n$ com o conjunto de nós V_S, um atributo de configuração de palavras é definido por uma função $(v \circ a)(c) : C \to Y$ composta por:

- uma função de endereçamento $a(c) : C \to V_S$
- uma função de atributo $v(w) : V_S \to Y$

Uma função de endereçamento pode, por sua vez, ser composta de funções mais simples, que operam em diferentes componentes da configuração de entrada c. Por exemplo:

- Funções que extraem a k-ésima palavra (partindo do topo) da pilha ou a k-ésima palavra (partindo da cabeça) do buffer.
- Funções que mapeiam uma palavra w para seu pai, filho mais à esquerda, filho mais à direita, irmão mais à esquerda ou irmão mais à direita no grafo de dependência definido por c.

Ao definir essas funções, podemos construir funções de endereçamento complexas que extraem, por exemplo, *"o irmão mais à direita do filho mais à esquerda do pai da palavra no topo da pilha"*, embora o fato de que as funções de endereçamento usadas na prática geralmente combinam no máximo três funções. Também vale a pena notar que a maioria das funções de endereçamento são parciais, o que significa que podem falhar ao retornar uma palavra. Por exemplo, uma função que deve retornar o filho mais à esquerda da palavra no topo da pilha é indefinida se a pilha estiver vazia ou se a palavra no topo da pilha não tiver filhos. Nesse caso, qualquer recurso definido com essa

função de endereçamento também será indefinido (ou terá um valor especial para indicar que é nulo).

Tipicamente, as funções de atributo referem-se a alguma propriedade linguística de palavras, que pode ser dada como entrada para o analisador ou computada como parte do processo de análise. Podemos exemplificar isso com a palavra "rapidamente" da sentença da Figura 2.4:

- Identidade de $w_i = rapidamente$
- Identidade do lemma de $w_i = rapidamente$
- Identidade da tag POS para $w_i = ADV$
- Identidade do rótulo de dependência para $w_i = amod$

Os três primeiros atributos são estáticos no sentido de que são constantes, se disponíveis, em todas as configurações para uma determinada sentença. Ou seja, se a sentença de entrada tiver sido lematizada e marcada com tags POS no pré-processamento, os valores desses recursos estarão disponíveis para todas as palavras da frase e seus valores não serão alterados durante a análise.

Por outro lado, o atributo de rótulo de dependência é dinâmico no sentido de que está disponível somente depois que o arco de dependência relevante foi adicionado ao conjunto de arcos. Assim, o rótulo de dependência para uma dada palavra dá sentença é indefinido nas as primeiras configurações, posteriormente terá o valor "amod" em todas as demais configurações. Assim, esses atributos podem ser usados para definir características do histórico de transição e da árvore de dependência parcialmente construída, o que acaba sendo uma das principais vantagens da abordagem baseada em transição.

Vamos agora examinar uma representação de atributos completa usando apenas atributos de configuração. A Tabela 3.1 mostra um modelo simples com sete atributos, cada um definido por uma função de endereço e uma função de atributo. Usamos a notação si e bi para referir-se a i-ésima palavra na pilha e no buffer, respectivamente[1], e usamos $leftj(w)$ e $rightj(w)$ para indicar o j-ésimo filho mais distante de w para a esquerda e para a direita, respectivamente. As funções de atributo usadas são FORM para a forma da palavra e DEPREL para label de dependência. O valor especial NULL é usado para indicar que um atributo é indefinido em uma determinada configuração.

Embora o template definido na Tabela 4 seja suficiente para construir um analisador funcional, geralmente é necessário um modelo mais complexo para obter uma boa precisão de análise. Na capítulo 4 é apresentado um modelo um

[1] Observe que a indexação começa em 0, de modo que s0 representa a palavra no topo da pilha, enquanto b0 é a primeira palavra no buffer.

Endereço	Atributo
s_0	FORM
b_0	FORM
b_1	FORM
$left_0(s_0)$	DEPREL
$right_0(s_0)$	DEPREL
$left_0(b_0)$	DEPREL
$right_0(b_0)$	DEPREL

Tabela 3.1: Exemplo de template de atributos.

pouco mais complexo e mais representativo dos sistemas de análise utilizados atualmente: o modelo utilizado em Chen e Manning 2014.

Assim definimos os chamados atributos de configuração, ou seja, atributos que podem ser definidos pela composição de uma função de endereço e uma função de atributo, uma vez que esses são os atributos mais importantes na análise baseada em transição. Em princípio, no entanto, os atributos podem ser definidos sobre quaisquer propriedades de uma configuração que se acredita serem importantes para prever a transição correta. Um tipo de atributo que tem sido usado com frequência é a distância entre duas palavras, normalmente a palavra no topo da pilha e a primeira palavra no buffer de entrada. Isso pode ser medido pelo número de palavras que intervêm, possivelmente restritas a palavras de um determinado tipo, como verbos. Outro tipo comum de atributo é o número de filhos de uma determinada palavra, possivelmente divididos em filhos à esquerda e filhos à direita.

3.5 Treino

Uma vez que tenhamos definido nossa representação de atributos, queremos aprender a prever a transição correta $o(c)$, para qualquer configuração c, dada a representação da configuração $\varphi(c)$ como entrada. Em termos de aprendizado de máquina, esse é um problema de classificação direto, em que as instâncias a serem classificadas são (representações de atributos de) configurações e as classes são as transições possíveis (conforme definido pelo sistema de transição). A figura 3.4 ilustra este cenário.

Em uma abordagem supervisionada, os dados de treinamento devem consistir em instâncias marcadas com a classe correta, o que significa que nossas instâncias de treinamento devem ter a forma $(\varphi(c), t)(t = o(c))$. No entanto, esta não é a forma em que os dados de treinamento estão diretamente disponíveis para nós em um banco de árvores.

Em um banco de árvores o conjunto de treino D é constituído sentenças

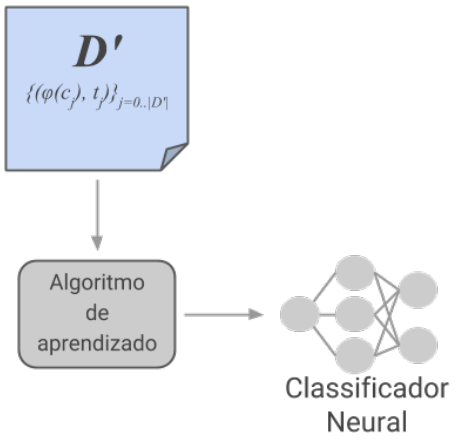


Figura 3.4: Treinamento de um classificador neural a partir de um conjunto $D^{'}$ contendo atributos de configurações e transições.

pareadas com suas respectivas árvores de dependência. Isso pode ser definido como:

$$D = (D_d, T_d)_{d=0..|D|}$$

A fim de treinar um classificador para análise de dependência baseada em transição, devemos, portanto, encontrar uma maneira de derivar a partir de D um novo conjunto de treinamento $D^{'}$, consistindo de configurações emparelhadas com suas transições corretas:

$$D^{'} = (\varphi(c_d), t_d)_{d=0..|D^{'}|}$$

Para derivar o conjunto $D^{'}$ utilizamos um componente capaz de prever qual transição correta aplicar em uma dada configuração para se chegar na árvore de dependência golden correspondente. Chamamos este componente de Oráculo, ilustrado na figura 3.5.

3.6 Oráculo

Construímos $D^{'}$ a partir de D da seguinte forma:

- Para cada instância $(S_d, G_d) \in D$, primeiramente deve-se construir uma sequência de transições $C_{0,md} = (c_0, c_1, ..., c_m)$ tal que:
  - $c_0 = c_0(S_d)$
  - $G_d = (V_d, A_{cm})$
- Para cada configuração não terminal $c_d^i \in C_{0,m}^d$, é adicionado a $D^{'}$ um par $(\varphi(c_i^d), t_i^d)$, onde $t_i^d(c_i^d) = c_{i+1}^d$.

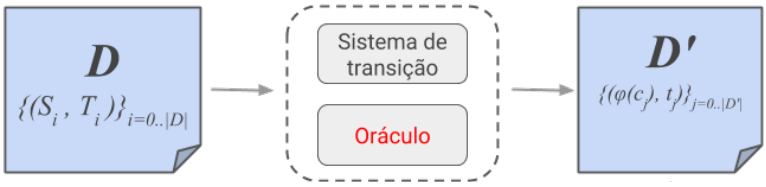


Figura 3.5: Utilizando-se o componente oráculo, convertemos o banco de árvores original em um conjunto contendo transições. Afim de treinar o classificador capaz de prever transições.

Este esquema pressupõe que, para cada sentença S_d com árvore de dependência G_d, podemos construir uma sequência de transições que resulta em G_d. Desde que todas as árvores de dependência sejam projetivas, podemos fazer isso usando o algoritmo 1 e confiando na árvore de dependência $G_d = (V_d, A_d)$ para computar a função oráculo na linha 3 da seguinte maneira:

$$o(c) = \begin{cases} LEFT, & \text{se } (\beta[0], r, \sigma[0]) \in A_d). \\ RIGHT, & \text{se } (\sigma[0], r, \beta[0]) \in A_d \text{ e } \forall w, r; (\beta[0], r, w) \in A_d \Rightarrow (\beta[0], r, w) \in A. \\ SHIFT, & \text{caso contrário.} \end{cases}$$

O primeiro caso afirma que a transição correta é $LEFT$ se a árvore de dependência correta tiver um arco da primeira palavra $\beta[0]$ no buffer de entrada para a palavra $\sigma[0]$ no topo da pilha com o rótulo de dependência r. O segundo caso afirma que a transição correta é $RIGHT$ se a árvore de dependência correta tiver um arco de $\sigma[0]$ a $\beta[0]$ com rótulo de dependência r. Mas somente se todos os arcos de saída de $\beta[0]$ (de acordo com G_d) já tiverem sido adicionados a A. A condição extra é necessária porque, após a transição $RIGHT$, a palavra $\beta[0]$ não estará mais na pilha e nem no buffer, o que significa que será impossível adicionar mais arcos envolvendo essa palavra. Nenhuma condição correspondente é necessária para o caso $LEFT$, uma vez que isso será satisfeito automaticamente, desde que a árvore de dependência correta seja projetiva. O terceiro e último caso cuida de todas as configurações restantes, onde $SHIFT$ deve necessariamente ser a transição correta, incluindo o caso especial em que a pilha está vazia.

Treinar um classificador no conjunto $D = (\varphi(c_d), t_d)_{d=0..|D|}$ é um problema padrão na análise de dependência com aprendizado de máquina. Que pode ser resolvido usando uma variedade de diferentes algoritmos de aprendizado supervisionado. Esta estratégia de treino é a mesma utilizada em Chen e Manning 2014, embora este utilize um sistema de transição diferente.

A figura 3.6 ilustra a arquitetura utilizada para obtermos o classificador partindo do conjunto original.

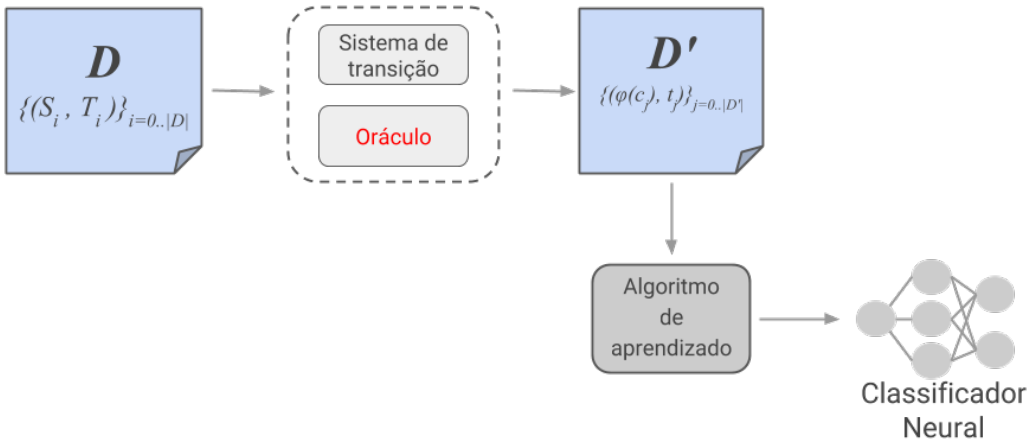


Figura 3.6: A partir do banco de árvores original utilizamos o componente oráculo em conjunto com um sistema de transição para obtermos um conjunto de treino para o classificador neural.

No final deste capítulo é apresentamos uma forma alternativa de treinamento que utiliza a técnica chamada de oráculo dinâmico (Goldberg e Nivre 2013).

3.7 Teste

A etapa de teste consiste em determinar a estrutura sintática correspondente a sentenças que não estavam no conjunto de treino. Depois que um modelo é obtido através do treino, isto é, o vetor de parâmetros W foi aprendido, fazer a análise de uma nova estrutura é simples. Isso pode ser feito utilizando o algoritmo determinístico de análise de dependência com classificador apresentado anteriormente. Agora reescrito com o intuito de enfatizar a utilização do modelo:

Algorithm 3: Algoritmo determinístico de análise de dependência com classificador. Parte 2.

input : S, T, W, φ
output: A_c
$c \leftarrow c_0(S)$;
while *c is not terminal* **do**
  $t_p \leftarrow argmax_{t \in LEGAL(c)} W \cdot \varphi(c, t)$;
  $c \leftarrow t_p(c)$;
end

Este processo pode ser feito para cada sentença de um conjunto de teste. Aplicando gulosamente a transição legal escolhida pelo classificador a cada passo até chegar a uma configuração terminal.

3.8 Variações do sistema

Até aqui, consideramos um único sistema de transição, e um único algoritmo de análise determinístico. No entanto, existem muitas variações possíveis no tema básico da análise baseada na transição, obtido pela variação do sistema de transição, o algoritmo de análise ou ambos. Além disso, existem muitos algoritmos de aprendizagem possíveis que podem ser usados para treinar classificadores, um tópico abordado na seção anterior. Nesta seção, introduziremos alguns sistemas de transição alternativos, algumas variações no algoritmo básico de análise, treino online e representação da entrada.

Uma das peculiaridades do sistema de transição definido anteriormente neste capítulo é que os dependentes à direita não podem ser ligados à sua cabeça até que todos os seus dependentes tenham sido anexados. Como conseqüência, pode haver incerteza sobre se uma transição $RIGHT$ é apropriada, mesmo se for certo que a primeira palavra no buffer de entrada deve ser dependente da palavra no topo da pilha. Este problema é eliminado em uma variação deste sistema de transição, chamada de *Arc-Eager*, definida a seguir.

3.8.1 O sistema Arc-Eager

Neste sistema, que é chamado *Arc-Eager* porque todos os arcos (apontando para a esquerda ou para a direita) são adicionados o mais rápido possível, a transição $RIGHT$ é redefinida para que a palavra dependente w_j seja empurrada para a pilha (em topo de sua cabeça w_i), tornando possível adicionar mais dependentes a essa palavra. Além disso, temos que adicionar uma nova transição $REDUCE$, que possibilita desempilhar a palavra dependente da pilha em um momento posterior, e que tem como pré-requisito que a palavra no topo da pilha já tenha uma cabeça , isto é, que o conjunto de arco contém um arco (w_k, r', w_i) para algum k e r' (onde w_i é a palavra no topo da pilha).

Vale ressaltar, que assim como o sistema definido anteriormente, este sistema também é sólido e completo em relação à classe de árvores de dependência projetiva (ou florestas que podem ser transformadas em árvores, para ser exato), e tem complexidade linear de tempo e espaço quando combinados com o algoritmo determinístico de análise formulado anteriormente (Nivre 2008). Como modelos de análise ($M = (T, \lambda, h)$), os dois sistemas são, por-

tanto, equivalentes em relação aos componentes T e h, mas diferem em relação ao componente λ, uma vez que os diferentes conjuntos de transição dão origem a diferentes parâmetros que precisam ser aprendidos a partir dos dados.

Concretamente, no sistema *Arc-Eager* (Nivre 2003), uma configuração $c = (\sigma, \beta, A)$ consiste em uma pilha σ, um buffer β, e um conjunto A de arcos de dependência. Dada uma sentença $S = w_1, ..., w_n$, o sistema é inicializado com uma pilha vazia, um conjunto de arco vazio e $\beta = w_1, ..., w_n, ROOT$, onde $ROOT$ é o nó raiz artificial. Qualquer configuração c com uma pilha vazia e um buffer contendo apenas $ROOT$ é considerada como terminal e a árvore de dependência resultante é obtida através do conjunto de arcos A_c de c.

O sistema tem 4 transições: $RIGHT, LEFT, SHIFT, REDUCE$, definidas da seguinte forma:

- $SHIFT[(\sigma, [b|\beta], A)] = ([\sigma|b], \beta, A)$
- $RIGHT[([\sigma|s], [b|\beta], A)] = ([\sigma|s|b], \beta, A \cup (s, r, b))$
- $LEFT[([\sigma|s], [b|\beta], A)] = (\sigma, [b|\beta], Acup(b, r, s))$
- $REDUCE[([\sigma|s], \beta, A)] = (\sigma, \beta, A)$

Há algumas pré-condições que determinam:

- as transições $RIGHT$ e $SHIFT$ são legais apenas quando $\beta = ROOT$.
- as transições $LEFT, RIGHT$ e $REDUCE$ são legais somente quando a pilha não estiver vazia.
- $LEFT$ só é legal quando s não tem um pai em A.
- $REDUCE$ só é legal quando s tem um pai em A.

O sistema *Arc-Eager* constrói árvores ansiosamente no sentido de que os arcos são adicionados o mais cedo possível. Além disso, cada palavra coletará todos os seus dependentes da esquerda antes de coletar seus dependentes corretos.

3.8.2 O sistema Arc-Standard

Sistema que é utilizado no parser do trabalho de Chen e Manning 2014. Originalmente definido em Nivre e Scholz 2004, este sistema tem configurações da forma $c = (\sigma, \beta, A)$ tal como o sistema Arc-Eager.

A configuração inicial para uma sentença $S = w_1, .., w_n$ tem uma pilha vazia e um conjunto de arcos vazio, e $\beta = ROOT, w_1, ..., w_n$. Uma configuração c é terminal se tiver um buffer vazio e uma pilha contendo o nó único $ROOT$. A árvore de dependência resultante é dada por A_c. O sistema possui 3 transições: $RIGHT, LEFT, SHIFT$, definidas da seguinte forma:

- $SHIFT[(\sigma, [b|\beta], A)] = ([\sigma|b], \beta, A)$
- $RIGHT[([\sigma|s_1|s_0], \beta, A)] = ([\sigma|s_1], \beta, A \cup (s_1, r, s_0))$
- $LEFT[([\sigma|s_1|s_0], \beta, A)] = (\sigma|s_0, \beta, A \cup (s_0, r, s_1))$

Com as seguintes restrições:

- $RIGHT$ é legal somente quando a pilha tiver pelo menos dois elementos.
- $LEFT$ é legal somente quando a pilha tiver pelo menos dois elementos e $s1 \neq ROOT$.

O sistema *Arc-Standard* constrói árvores de uma forma *bottom-up*: cada palavra deve coletar todos os seus dependentes antes de ser anexada à sua cabeça. O sistema não apresenta restrição quanto à ordem de coleta de dependentes de esquerda e direita.

Analogamente ao sistema Arc-Eager, este sistema suporta apenas árvores de dependência projetivas.

3.8.3 Árvores não projetivas

Os sistemas de transição apresentados anteriormente são capazes de derivar apenas árvores projetivas. Nesta seção é apresentada uma forma de superar essa limitação adicionando uma nova transição ao sistema Arc-Standard.

O novo sistema, proposto inicialmente em Nivre 2009, usa uma transição chamada $SWAP$ para reorganizar os nós movendo o segundo nó da pilha de volta para o buffer (para que o nó no topo da pilha seja o mesmo antes e depois da transição). Este sistema tem configurações da forma $c = (\sigma, \beta, A)$ tal como o sistema *Arc-Standard*.

A configuração inicial para uma sentença $S = w_1, ..., w_n$ tem uma pilha com apenas o elemento raiz $ROOT$, um conjunto de arcos vazio, e $\beta = w_1, ..., w_n$. Uma configuração c é terminal se tiver um buffer vazio e uma pilha contendo o nó único $ROOT$; a árvore de dependência resultante é dada por A_c. O sistema possui 4 transições: $RIGHT, LEFT, SHIFT$, definidas tal como no sistema *Arc-Standard* e a transição adicional $SWAP$:

- $SHIFT[(\sigma, [i|\beta], A)] = ([\sigma|i], \beta, A)$
- $RIGHT[([\sigma|i|j], \beta, A)] = ([\sigma|i], \beta, A \cup (i, r, j))$
- $LEFT[([\sigma|i|j], \beta, A)] = (\sigma|j, \beta, Acup(j, r, i))$
- $SWAP[([\sigma|i|j], \beta, A)] = ([\sigma|j], [i|\beta], A)$

Com as seguintes restrições:

- $LEFT$ é permitida somente quando $s1 \neq ROOT$.
- $LEFT$ e $RIGHT$ são legais somente quando a pilha tiver pelo menos dois elementos.
- $SWAP$ é permitida somente quanto $i \neq 0$ e $i < j$.

A transição $SWAP$ não é permitida se o nó i a ser movido de volta for o nó raiz artificial ou se os dois nós i e j já tiverem sido trocados. É possível mostrar que a transição $SWAP$ nos permite fazer permutações arbitrárias da ordem original das palavras (constituindo um algoritmo de ordenação simples), o que significa que sempre podemos encontrar uma reordenação que torne a árvore de destino projetiva. Demonstrou-se que essa técnica fornece boa precisão para análise de dependência não-projetiva, especialmente em combinação com busca em feixe e aprendizado estruturado Bohnet e Nivre 2012.

3.8.4
Análise não determinística

Até aqui, o oráculo foi modelado de uma forma simplificada no sentido de que ele considera um único caminho de transições até a árvore de dependência correta. O que impacta negativamente na performance do modelo. Uma das razões disso é a propagação de erro: uma vez que o classificador cometa um erro, mais erros provavelmente ocorrerão. Esse comportamento está intimamente relacionado à maneira como os analisadores gulosos são normalmente treinados. Dado um banco de árvores, uma sequencia de transições corretas (*golden*) é derivada, e um preditor é treinado para prever transições ao longo desta sequencia, sem considerar qualquer estado do analisador que esteja fora dessa

sequência. Assim, uma vez que o analisador se desvie do caminho de transições corretas durante o teste, ele se aventura em território desconhecido e é forçado a reagir a situações para as quais nunca foi treinado. De fato, há situações em que mais de uma transição pode levar à árvore de dependência correta. Os algoritmos de análise descritos aqui executam uma busca gulosa e determinística da sequência de transição ideal, explorando apenas uma única sequência de transição e terminando assim que atinge uma configuração terminal. Dado um dos sistemas de transição descritos até agora, isso acontece após uma única passagem da esquerda para a direita sobre as palavras da sentença de entrada.

Uma alternativa a essa estratégia é relaxar a hipótese do determinismo e explorar mais de uma sequência de transição em uma única passagem. A maneira mais simples de fazer isso é usar a busca por feixe (*beam-search*), ou seja, manter as k sequencias de transição parciais mais promissoras após cada etapa de transição. Isso requer que tenhamos uma maneira de pontuar e classificar todas as transições possíveis de uma determinada configuração, o que significa que a aprendizagem não pode mais ser reduzida a um problema de classificação pura. Além disso, precisamos de uma maneira de combinar as pontuações para transições individuais de tal forma que possamos comparar as sequências de transição que podem ou não ser do mesmo tamanho, o que é um problema não trivial para a análise de dependência baseada em transição. No entanto, desde que o tamanho do feixe seja limitado por uma constante k, o pior tempo de execução ainda é $O(n)$. Trabalhos que implementam esta abordagem incluem Straka, Hajic et al. 2015 e Weiss et al. 2015.

O trabalho de Goldberg e Nivre 2013 propõe uma alternativa mais simples utilizando oráculos dinâmicos, que permitem explorar dinamicamente caminhos alternativos e não ideais durante o treino.

3.8.5 Oráculo dinâmico

Uma forma simples de treinar o analisador em caminhos alternativos ao caminho ótimo é através de uma técnica chamada de oráculo dinâmico. Nesta seção é apresentada uma forma de fazer isso e ainda eliminar a necessidade de se gerar um conjunto de treinamento adicional, como mostrado anteriormente.

A ideia é iterar sobre o conjunto de dados original usando uma função oráculo $o(c, T)$ para orientar a convergência do modelo. A função oráculo $o(c, T)$ ao invés de retornar a transição ótima, como definido anteriormente, retorna um conjunto de transições. Definidas como corretas, ou seja, transições de custo zero dada uma configuração c e uma árvore de análise T. Esta técnica

é baseada no trabalho de Goldberg e Nivre 2013. O algoritmo 4 mostra como treinar um modelo online usando o oráculo dinâmico.

Algorithm 4: Treinamento online de um analisador guloso.

```
   input  : Dataset, φ
   output: Model
 1 W ← distribuição aleatória;
 2 for S, T in Dataset do
 3 |   c ← c_0(S);
 4 |   while c is not terminal do
 5 |   |   CORRECT(c) ← o(c, T);
 6 |   |   t_p ← argmax_{t∈LEGAL(c)} W · φ(c, t);
 7 |   |   t_o ← argmax_{t∈CORRECT(c)} W · φ(c, t);
 8 |   |   if t_p not in CORRECT(c) then
 9 |   |   |   UPDATE(W, φ(c, t_0), φ(c, t_p));
10 |   |   |   c ← t_o(c);
11 |   |   else
12 |   |   |   c ← t_p(c);
13 |   |   end
14 |   end
15 end
```

Usamos $CORRECT(c)$ para nos referirmos ao conjunto de transições que são possíveis em uma configuração c. Depois de predizer a próxima transição t_p usando o modelo na linha 6 nós verificamos se ela está contida no conjunto $CORRECT(c)$ das transições que são consideradas corretas pelo oráculo (linhas 5 e 8). Se a transição prevista t_p não estiver correta, atualizamos os parâmetros do modelo W afastando-se de t_p e em direção à previsão do oráculo para o que é a transição de maior pontuação no modelo atual. E passamos para a próxima configuração (linhas 8–10). Se t_p estiver correto, simplesmente o aplicamos e movemos para $t_p(c)$ sem alterar os parâmetros do modelo (linha 12).

4
Experimentos

Este capítulo descreve a avaliação empírica realizada em um sistema baseado em transição implementado com base na modelagem especificada em Chen e Manning 2014, que estabeleceu o estado-da-arte no ano de sua publicação.

Sempre tentando tirar proveito do setup experimental provido pelo conjunto de bancos de árvores disponibilizados pela iniciativa *Universal Dependencies* juntamente com a formalização de tarefas, métricas de avaliação e transparência com respeito aos sistemas participantes nas conferências CoNLL. O que permite replicar e reproduzir experimentos com uma base sólida para avaliação. De fato, um dos objetivos dessas iniciativas é estimular que abordagens de aprendizado de máquina adotem um padrão para avaliar seus desempenhos.

Aqui, nosso objetivo é avaliar empiricamente a modelagem baseada em transição. Para conseguir isso, propomos testar nosso esquema usando o conjunto de dados das línguas portuguesa e inglesa disponibilizados através da iniciativa UD, tal como na tarefa compartilhada da CoNLL de 2017. Bem como uma das métricas padrão propostas nessa competição. Este capítulo apresenta a configuração experimental, as estatísticas dos dados e a performance de nossa abordagem de aprendizado de máquina proposta para análise de dependência.

Em primeiro lugar, os conjuntos de dados são descritos e analisados, bem como a métrica de avaliação. Em seguida, a aplicação da solução de aprendizado de máquina proposta é descrita. Finalmente, relatamos o desempenho alcançado nos experimentos, suas particularidades de modelagem e configuração de parâmetros específicos.

4.1
Modelagem

Aqui nosso analisador é modelado de forma a utilizar um classificador de rede neural FFNN e um algoritmo de análise guloso com um sistema baseado em transição *Arc-Standard* com *SWAP*. Caracterizando assim uma adaptação do trabalho de Chen e Manning 2014.

4.1.1 Algoritmo de transição

A modelagem original proposta em Chen e Manning 2014 utiliza o sistema de transição *Arc-Standard*, que é aplicável apenas à análise de árvores projetivas. Mas considerando que os bancos de árvores no dataset utilizado em nossos experimentos contém sentenças não projetivas nos conjuntos de treino e teste, optou-se por utilizar a variante que inclui a transição $SWAP$ do sistema original, conforme descrito na seção 3.8.3.

A complexidade do pior caso deste algoritmo é $O(n^2)$, mas na prática acaba sendo linear uma vez que as poucas árvores não projetivas encontradas no corpus (cerca de 5% do total).

4.1.2 Classificador

Na arquitetura deste classificador neural as entradas são primeiro transformadas por uma camada com unidades lineares retificadas (ReLU). Em seguida, por uma camada oculta semelhante a camada de entrada. Que é passada como entrada para uma camada softmax, cuja saída atribui uma previsão de probabilidade para cada transição da máquina. A Figura 4.1 apresenta uma ilustração desta arquitetura.

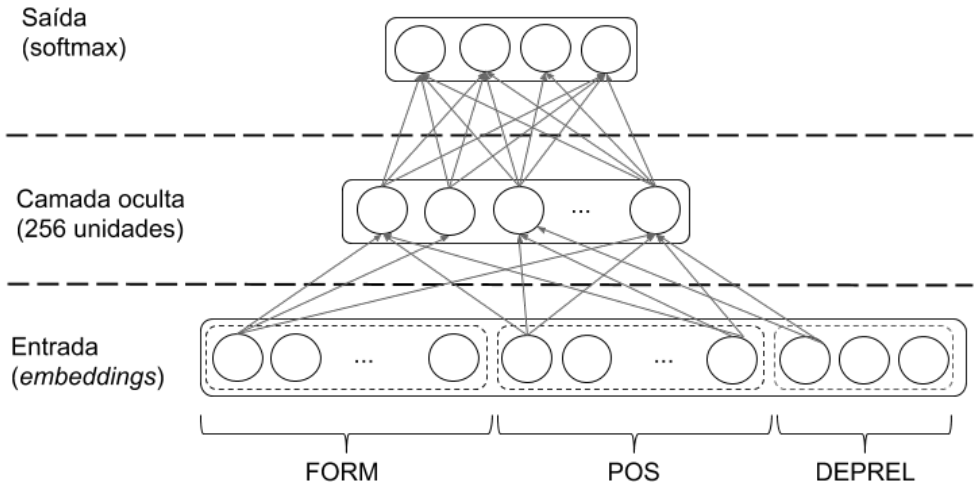


Figura 4.1: Arquitetura da rede.

Os pesos para todas as camadas são inicializados em um distribuição uniforme aleatória. A camada oculta ReLU tem seu viés inicializado com 1. Nos experimentos também mostrou-se útil a aplicação da técnica de regularização

Dropout para a camada de entrada e para a camada oculta. A rede é treinada através de backpropagation utilizando o algoritmo AdaMax (Kingma e Ba 2014).

4.1.3 Atributos

Uma das vantagens de se utilizar uma rede neural como classificador é o fato deste modelo não requerer uma engenharia de atributos. Bastando alguns atributos atômicos de informações disponíveis localmente, isto é, informações sobre o estado e o histórico do sistema de transição. Que chamados de atributos de configuração.

Selecionamos o mesmo conjunto de atributos utilizados em Chen e Manning 2014). Nesta seção definimos tais atributos através da composição de uma função de endereço e uma função de atributo. Assim como no capítulo 3 usaremos aqui a notação s_i e b_i para referir-se a i-ésima palavra na pilha e no buffer, respectivamente. E usamos $left_j(w)$ e $right_j(w)$ para indicar o j-ésimo filho mais distante de w para a esquerda e para a direita, respectivamente. Nos experimentos realizados utilizamos um total de 18 elementos. Concretamente:

- Os três primeiros elementos do topo da pilha
  - s_0, s_1, s_2
- Os três elementos do início do buffer
  - b_0, b_1, b_2
- O primeiro e segundo filho mais a esquerda e primeiro e segundo filho mais a direita das duas palavras do topo da pilha
  - $left_1(s_1), left_2(s_1), left_1(s_2), left_2(s_2)$
  - $right_1(s_1), right_2(s_1), right_1(s_2), right_2(s_2)$
- O primeiro e segundo filhos mais à esquerda do mais à esquerda e mais à direita do mais à direita das duas primeiras palavras na pilha
  - $left_1(left_1(s_1)), left_1(left_1(s_2))$
  - $right_1(right_1(s_1)), right_1(right_1(s_2))$

Para cada elemento, utilizamos os campos FORM, UPOSTAG, e DEPREL. Cada campo de um elemento é representado através de um ponto em um espaço vetorial real com d dimensões. Sendo 50 dimensões para o campo FORM, 12 para representar o campo UPOSTAG e 16 para representar o campo DEPREL.

4.2 Setup experimental

Nossa estratégia é tomar como baseline uma modelagem com camada de entrada inicializada com uma distribuição aleatória. Em seguida evoluir o sistema substituindo a distribuição aleatória por valores obtidos com pré treino utilizando a ferramenta *Word2vec* sobre o conjunto de treino aumentado com mais texto puro.

O ambiente preparado para realizar os experimentos tem como objetivo permitir testar os componentes de aprendizado de máquina da modelagem proposta com diferentes conjuntos de dados, avaliando sua performance de forma confiável e minimizando o custo de implementação. Podemos listar algumas características deste ambiente, que foi elaborado para permitir:

- Experimentos com diferentes conjunto de dados, neste caso os conjuntos das língua portuguesa e inglesa.
- O acompanhamento da evolução da performance do sistema no treino de forma confiável.
- Alterar a arquitetura da rede neural do classificador de forma declarativa.

Para montar este setup diversos recursos foram utilizados, sendo os recursos de software gratuitos e de código aberto. Nesta seção serão abordados aspectos destes recursos em mais detalhes. A começar pelo corpus, passando pelos recursos de software e hardware utilizados.

4.2.1 Dataset

Análise de dependência foi tarefa compartilhada em diversas conferencias CoNLL, incluindo as edições de 2017 e 2018. Na edição de 2017, foram disponibilizados conjuntos de treino para 45 línguas através do release UD 2.0. Uma descrição de como esse conjunto foi gerado pode ser encontrada em Mcdonald et al. 2013.

Os experimentos descritos neste capítulo utilizam as línguas inglesa e portuguesa presentes neste release (referente aos códigos en e pt_br no conjunto).

Este corpus segue o formato CoNLL-U, que contém dez atributos anotados para cada palavra. A saber: forma da palavra, lema da palavra, tag POS universal, tag POS específico da linguagem, lista de características morfológicas com valor agregado, cabeça sintática, rótulo sintático, grafo de dependência e um atributo reservado para anotações diversas.

As Tabelas 4.1 e 4.2 fornecem informações estatísticas sobre os conjuntos de treino utilizados nos experimentos.

Português (conjunto de treino "pt_br")	
Número de sentenças	10874
Número de símbolos (tokens)	268124
Tokens por sentença	24,6
Composições[1]	19760
Sentenças não- projetivas (%)	5,8

Tabela 4.1: Estatísticas do conjunto de treino da língua portuguesa do Brasil.

Inglês (conjunto de treino "en")	
Número de sentenças	14545
Número de símbolos (tokens)	229753
Tokens por sentença	15,8
Composições	0
Sentenças não- projetivas (%)	5,2

Tabela 4.2: Estatísticas do conjunto de treino da língua portuguesa do Brasil.

Uma parte dos dados de treinamento correspondente a aproximadamente 10% do total de cada conjunto de treino foi utilizada para efeito de validação. Os conjuntos de teste utilizados foram os mesmos conjuntos de teste da tarefa compartilhada na CoNLL 2017. A Tabela 4.3 apresenta o tamanho de cada conjunto.

Corpus	**Treino**	**Validação**	**Teste**
Inglês (en)	12543	2002	2077
Português (pt_br)	9664	1210	1204

Tabela 4.3: Número de sentenças nos conjuntos de treino, validação e teste.

4.2.2 Implementação

Dado que o foco destes experimentos é avaliar o componente de aprendizado de máquina do sistema, isto é, o classificador, utilizamos implementações prontas de um sistema de transição e oráculo. Tirando proveito da ampla oferta de recursos de software disponíveis de forma gratuita e de código aberto que implementam sistemas de análise de dependência baseados em transição. A implementação desses componentes foi obtida de Yu et al. 2017, que publicou um sistema baseado em transição na forma de código aberto em linguagem Python.

O classificador também foi desenvolvido de forma a minimizar o custo de implementação e ao mesmo tempo facilitar mudanças na arquitetura, como ajuste do número de unidades ocultas. Por isso foi implementado utilizando Keras com Theano de Bergstra et al. 2010.

4.2.3 Preparação dos dados

Um conjunto de treino contendo pares do tipo (configuração, transição) é necessário para treinar nosso classificador. Mas dado que o dataset original não possui o conjunto de transições, mas sim árvores, exemplos de treino tiveram que ser gerados artificialmente. Para isso utilizamos o esquema proposto na seção 3.5, ilustrado na Figura 3.5.

Tendo em vista a comparação direta dos resultados com o estado-da-arte, na etapa de treino partimos de conjuntos com segmentação e tags POS obtidas da utilização do *UDPipe* ao invés de utilizar o conjunto anotado manualmente (*golden*). Ou seja, treinamos o sistema em um conjunto *silver*. Pois estes conjuntos foram os mesmos conjuntos utilizados pelos participantes da CoNLL-2017, sobre os quais foram publicados os resultados oficiais.

4.2.4 Treino e Validação

A representação da entrada foi obtida utilizando-se a ferramenta *Word2vec* de Mikolov et al. 2013, treinado nas palavras do conjunto de treino com 50 dimensões para o campo FORM. E vetores inicializados aleatóriamente para os campos UPOSTAG e DEPREL.

Tomamos como hiper-parâmetros o número de épocas de treino, a taxa de dropout e numero de unidades na camada oculta. Para calibrá-los utilizamos avaliação empírica treinando em 10% do conjunto de treino. Sumarizados na tabela 4.4.

Hiper-parâmetro	Valores testados	Valor escolhido
Épocas de treino	1 a 64	32
Número de unidades ocultas	128, 256, 300	256
Taxa de droput	20%, 25%, 50%	20%

Tabela 4.4: Escolha dos hiper-parâmetros treinando em 10% do conjunto de treino.

Os valores foram escolhidos com base em uma relação entre performance e eficiência. Observando-se a evolução da performance no conjunto de validação ao decorrer de 64 épocas de treino, verificou-se que após 32 épocas não houve melhoria de performance. Pois quando há ganho de performance no conjunto

de validação após a trigésima segunda época tal ganho é nulo ou então muito pequeno quando levamos em conta o esforço computacional exigido. Além disso, o modelo ótimo se continuar sendo treinado pode passar a um estado de *overfit*. Sendo assim fixamos o algoritmo para treinar por 32 épocas. O numero de unidades ocultas e taxa de dropout que obteve melhor relação entre performance e eficiência foi de 256 e 20% respectivamente.

O treinamento no conjunto completo foi realizado em uma máquina virtual com 2 CPUs e 8GB de memória. Levando aproximadamente 8hs para executar.

4.2.5 Teste

Os testes foram realizados no conjunto de teste anotado com o *UDPipe* (*silver*). Deixando o conjunto de teste com tags humanamente anotadas (*golden*) reservado para realizar a avaliação final. Estratégia que visa tornar este trabalho diretamente comparável com o estado-da-arte. Utilizando inclusive o mesmo script de validação.

4.2.6 Métrica de avaliação

A avaliação dos experimentos aqui descritos utiliza a métrica LAS da forma como definida na tarefa compartilhada pela CONLL 2017. Na prática utilizando o mesmo script de validação[2] que serviu para medir a performance dos sistemas participantes na conferência. Com isso se tem uma forma confiável, padronizada e de código livre para efetuar as avaliações empíricas.

Como mencionado anteriormente na seção 2.9, são duas as principais métricas de avaliação utilizadas atualmente para avaliar a performance dos sistemas de análise de dependência: UAS e LAS. Ambas medem a qualidade do sistema pela similaridade entre as árvores de dependência que ele produz e as árvores de dependência anotadas correspondentes.

Seguindo a estratégia utilizada na CoNLL-2017, temos que uma dependência é pontuada como correta apenas se ambos os nós da relação corresponderem aos nós anotados existentes. E as métricas UAS/LAS são calculadas através da média harmônica entre Precisão P e Recall R: $F_1 = 2PR/(P+R)$. Sendo a precisão P dada pelo número de relações corretas dividido pelo número total de nós, ambos produzidos pelo analisador a ser avaliado. E Recall R

[2] http://universaldependencies.org/conll17/evaluation.html

o número de relações corretas produzidas pelo analisador dividido pelo número de nós corretos que constam no dataset anotado.

4.3 Resultados

Primeiramente utilizamos os campos FORM, DEPREL e UPOSTAG da camada de entrada inicializados com uma distribuição aleatória como baseline. Ao utilizar pré treino com *Word2vec* para inicializar o campo FORM, observou-se uma melhoria na performance. Ilustrada na tabela 4.3. Que mostra a performance alcançada em cada experimento em relação ao melhor resultado de um sistema baseado em transição neste corpus.

Sistema	**Descrição**	**en**	**pt_br**
HIT-SCIR	Melhor resultado	**79,94**	**88,71**
1 + pré treino	Uma camada oculta e pré treino	75,19	84,51
1 (*baseline*)	Uma camada oculta	75,08	83,97

Tabela 4.5: Métricas LAS obtidas partindo do baseline, que consiste na modelagem com uma única camada oculta e representação das palavras inicializada aleatoriamente. A performance do sistema aumenta quando a representação das palavras é obtida com pré treino e adicionando uma camada oculta adicional à rede.

O melhor resultado alcança a performance de 84,51 LAS no conjunto *pt_br* e 75,19 LAS no conjunto *en*. A seguir são aprezentadas algumas considerações sobre os resultados obtidos e idéias para futuras melhorias.

5
Conclusão

Com este trabalho mostramos a modelagem e funcionamento de um sistema baseado em transição para o problema da análise de dependência (*Transition-based Parsing*).

Vimos como essa formalização é atraente, especialmente tratando-se de análise projetiva. Uma vez que o número de operações necessárias para construir qualquer árvore de análise projetiva no sistema baseado em transição é linear no comprimento da sentença. Como o número de árvores projetivas nos datasets de análise de dependência é geralmente pequeno, a modelagem baseada em transição se mostra computacionalmente eficiente quando comparada a outros formalismos, como por exemplo os baseados em grafos.

Observa-se também que para um analisador projetado desta forma, as sentenças da língua inglesa são mais difíceis de se analisar que sentenças da língua portuguesa. Isso aliado ao fato de o *treebank* utilizado para a língua inglesa ser maior e mais projetivo que o da língua portuguesa sugere que esta última seja uma língua menos ambígua. De fato, algumas línguas são mais difíceis de se lidar computacionalmente.

A iniciativa UD é relativamente nova quando comparada a outros bancos de árvores, como por exemplo *Penn Treebank*. Isso reflete na qualidade dos sistemas: aqueles que são obtidos com UD apresentam performance inferior à performance de sistemas obtidos utilizando *Penn Treebank*. Mas essa diferença tende a diminuir com o avanço da iniciativa UD. Além disso, o conjunto de teste utilizado neste trabalho é um conjunto anotado com *silver tags* obtidas através do sistema *UDPipe*. Apresentando portanto erros de segmentação, tokenização e tags que se propagam e impactam negativamente na qualidade da análise. Já o conjunto utilizado para testes dos trabalhos que utilizam *Penn Treebank* é geralmente um conjunto anotado por humanos, portanto *golden*. O que contribui para a diferença de performance entre sistemas avaliados em UD e sistemas avaliados em *Penn Treebank*.

5.0.1 Trabalhos futuros

Embora os resultados estejam cerca de 4 pontos percentuais abaixo do melhor resultado de um sistema baseado em transição, a simplicidade do modelo aqui apresentado confere margem para melhorias. Como a substituição do algoritmo guloso por um mais elaborado, de modo a efetuar a análise não determinística como mostrado em 3.8.4.

Também pode-se melhorar a performance utilizando *word embeddings* pré treinados em conjuntos maiores e aumentando do tamanho da rede neural que compõe o classificador com uma taxa de *dropout* mais alta. Esta estratégia foi rejeitada devido ao hardware limitado disponível para realização dos experimentos conduzidos aqui. Já que ela resultaria no aumento da quantidade de parâmetros da rede e consequentemente nos recursos computacionais necessários para executar os experimentos.

Além disso, sugerimos uma continuação deste trabalho com uma modelagem seguindo a arquitetura do sistema HIT-SCIR (Che et al. 2017). Seguindo a estratégia aqui apresentada, mas desta vez utilizando recursos disponibilizados na edição de 2018 da CoNLL. Na qual o HIT-SCIR apresentou o melhor resultado LAS geral dentre os sistemas participantes, superando Stanford.

Outro aspecto que merece ser investigado é a forma de lidar com a ambiguidade. Característica desafiadora presente nas linguagens naturais. Neste trabalho, utilizamos apenas um vetor para representar o significado de uma palavra. Apesar do fato de que uma palavra pode assumir diferentes significados dependendo do contexto.

Por fim, técnicas que permitam pré-treinar todo o modelo com representações hierárquicas. Ao invés de inicializar apenas a primeira camada da rede neural, como mostramos neste trabalho, estão surgindo em diversas tarefas do NLP. Experimentos com essas técnicas podem revelar novas oportunidades de melhorias.

Referências Bibliográficas

Barroso, Yanely Milanés (2016). "Structured Learning with Incremental Feature Induction and Selection for Portuguese Dependency Parsing". Diss. de mestrado. PUC-Rio.

Bergstra, James et al. (2010). "Theano: A CPU and GPU math compiler in Python". Em: *Proc. 9th Python in Science Conf.* Vol. 1.

Bohnet, Bernd e Jonas Kuhn (2012). "The best of both worlds: a graph-based completion model for transition-based parsers". Em: *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, pp. 77–87.

Bohnet, Bernd e Joakim Nivre (2012). "A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing". Em: *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Association for Computational Linguistics, pp. 1455–1465.

Buchholz, Sabine e Erwin Marsi (2006). "CoNLL-X Shared Task on Multilingual Dependency Parsing". Em: *Proceedings of the Tenth Conference on Computational Natural Language Learning, CoNLL 2006, New York City, USA, June 8-9, 2006*, pp. 149–164. URL: http://aclweb.org/anthology/W/W06/W06-2920.pdf.

Che, Wanxiang et al. (2017). "The HIT-SCIR system for end-to-end parsing of universal dependencies". Em: *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pp. 52–62.

Chen, Danqi e Christopher Manning (2014). "A fast and accurate dependency parser using neural networks". Em: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 740–750.

Culotta, Aron e Jeffrey Sorensen (2004). "Dependency tree kernels for relation extraction". Em: *Proceedings of the 42nd annual meeting on association for computational linguistics*. Association for Computational Linguistics, p. 423.

De Marneffe, Marie-Catherine et al. (2014). "Universal Stanford dependencies: A cross-linguistic typology." Em: *LREC*. Vol. 14, pp. 4585–4592.

Ding, Yuan e Martha Palmer (2004). "Synchronous dependency insertion grammars: A grammar formalism for syntax based statistical MT". Em: *Proceedings of the Workshop on Recent Advances in Dependency Grammar*.

Dyer, Chris et al. (2015). "Transition-based dependency parsing with stack long short-term memory". Em: *arXiv preprint arXiv:1505.08075*.

Eisner, Jason M (1996). "Three new probabilistic models for dependency parsing: An exploration". Em: *Proceedings of the 16th conference on Computational linguistics-Volume 1*. Association for Computational Linguistics, pp. 340–345.

Goldberg, Yoav e Joakim Nivre (2013). "Training deterministic parsers with non-deterministic oracles". Em: *Transactions of the Association of Computational Linguistics* 1, pp. 403–414.

Huang, Liang e Kenji Sagae (2010). "Dynamic Programming for Linear-Time Incremental Parsing". Em: *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden*, pp. 1077–1086. URL: http://www.aclweb.org/anthology/P10-1110.

Kingma, Diederik P e Jimmy Ba (2014). "Adam: A method for stochastic optimization". Em: *arXiv preprint arXiv:1412.6980*.

Kiperwasser, Eliyahu e Yoav Goldberg (2016). "Easy-first dependency parsing with hierarchical tree LSTMs". Em: *arXiv preprint arXiv:1603.00375*.

Koo, Terry e Michael Collins (2010). "Efficient Third-Order Dependency Parsers". Em: *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden*, pp. 1–11. URL: http://www.aclweb.org/anthology/P10-1001.

Kuhlmann, Marco, Carlos Gómez-Rodrıguez e Giorgio Satta (2011). "Dynamic Programming Algorithms for Transition-Based Dependency Parsers". Em: *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*, pp. 673–682. URL: http://www.aclweb.org/anthology/P11-1068.

McDonald, Ryan T. e Fernando C. N. Pereira (2006). "Online Learning of Approximate Dependency Parsing Algorithms". Em: *EACL 2006, 11st Conference of the European Chapter of the Association for Computational*

Linguistics, Proceedings of the Conference, April 3-7, 2006, Trento, Italy. URL: http://aclweb.org/anthology/E/E06/E06-1011.pdf.

McDonald, Ryan T., Fernando Pereira et al. (2005). "Non-Projective Dependency Parsing using Spanning Tree Algorithms". Em: *HLT/EMNLP 2005, Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference, 6-8 October 2005, Vancouver, British Columbia, Canada*, pp. 523–530. URL: http://aclweb.org/anthology/H/H05/H05-1066.pdf.

Mcdonald, Ryan et al. (2013). "Universal dependency annotation for multilingual parsing". Em: *In Proc. of ACL '13*.

Mikolov, Tomas et al. (2013). "Efficient estimation of word representations in vector space". Em: *arXiv preprint arXiv:1301.3781*.

Nivre, Joakim (2003). "An efficient algorithm for projective dependency parsing". Em: *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*. Citeseer.

— (2006). *Inductive dependency parsing*. Springer.

— (2008). "Algorithms for deterministic incremental dependency parsing". Em: *Computational Linguistics* 34.4, pp. 513–553.

— (2009). "Non-projective dependency parsing in expected linear time". Em: *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*. Association for Computational Linguistics, pp. 351–359.

— (2015). "Towards a universal grammar for natural language processing". Em: *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer, pp. 3–16.

Nivre, Joakim e Ryan McDonald (2008). "Integrating graph-based and transition-based dependency parsers". Em: *Proceedings of ACL-08: HLT*, pp. 950–958.

Nivre, Joakim e Mario Scholz (2004). "Deterministic Dependency Parsing of English Text". Em: *COLING 2004, 20th International Conference on Computational Linguistics, Proceedings of the Conference, 23-27 August 2004, Geneva, Switzerland*. URL: http://www.aclweb.org/anthology/C04-1010.

Straka, Milan, Jan Hajic et al. (2015). "Parsing universal dependency treebanks using neural networks and search-based oracle". Em: *International Workshop on Treebanks and Linguistic Theories (TLT14)*, pp. 208–220.

Straka, Milan e Jana Straková (2017). "Tokenizing, pos tagging, lemmatizing and parsing ud 2.0 with udpipe". Em: *Proceedings of the CoNLL 2017*

Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies, pp. 88–99.

Tesnière, Lucien (1959). *Eléments de Syntaxe Structurale*. Paris: Klincksieck.

Wang, Mengqiu, Noah A. Smith e Teruko Mitamura (2007). "What is the Jeopardy Model? A Quasi-Synchronous Grammar for QA". Em: *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, pp. 22–32. URL: http://www.aclweb.org/anthology/D07-1003.

Weiss, David et al. (2015). "Structured training for neural network transition-based parsing". Em: *arXiv preprint arXiv:1506.06158*.

Yamada, Hiroyasu e Yuji Matsumoto (2003). "Statistical dependency analysis with support vector machines". Em: *Proceedings of IWPT*. Vol. 3. Nancy, France, pp. 195–206.

Yu, Kuan et al. (2017). "The parse is darc and full of errors: Universal dependency parsing with transition-based and graph-based algorithms". Em: *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies, Vancouver, Canada, August 3-4, 2017*, pp. 126–133. DOI: 10.18653/v1/K17-3013. URL: https://doi.org/10.18653/v1/K17-3013.

Zhang, Yue e Stephen Clark (2008). "A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search". Em: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, pp. 562–571.

Zhang, Yue e Joakim Nivre (2011). "Transition-based Dependency Parsing with Rich Non-local Features". Em: *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA - Short Papers*, pp. 188–193. URL: http://www.aclweb.org/anthology/P11-2033.