DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Automated Extraction of Causes and Effects from Natural Language Requirements by Fine-Tuning Tree Recursive Neural Networks

Noah Jadallah

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Automated Extraction of Causes and Effects from Natural Language Requirements by Fine-Tuning Tree Recursive Neural Networks

Automatische Extraktion von Ursachen und Effekten aus natürlichsprachlichen Anforderungen durch die Optimierung von Tree Recursive Neural Networks

Author:	Noah Jadallah
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Maximilian Junker, M.Sc. Jannik Fischbach
Submission Date:	18.08.2021

I confirm that this bachelor's thesis in information systems is my own work and I have documented all sources and material used.

Munich, 18.08.2021 Noah Jadallah

Acknowledgments

I would like to take this opportunity to thank my thesis advisor M.Sc. Jannik Fischbach for constantly giving me his advice and expertise. I would also like to thank my other advisor, Dr. Maximilian Junker, for his support, and for introducing me to the exciting world of Natural Language Processing.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Abstract

Requirements engineering and testing is a crucial part of the development process. However, deriving test cases from software requirements can be a time-consuming and error-prone task. To save development time and costs, automated causality extraction can be leveraged to automatically or semi-automatically build test cases from functional software requirements while checking for redundancy, dependency or contradictions between those requirements. This thesis aims to solve the problem of automated causality extraction for software requirements by making three contributions: We (1) experiment with various settings to fine-tune Recursive Neural Networks (RNN) for this task, (2) explore ways to improve inference of RNNs (3), introduce a tool to parse the composition of a causal relation as a tree structure.

(1) We conduct several experiments in order to fine-tune and optimize our model. There are three parameters that we change in our experiments. First, we train our model with two different datasets that are either built on the concepts of left-branching or right-branching trees. Second, we change the dimensions of the word embeddings vector. And third, we additionally train our model with pre-trained non-contextual and contextual word embeddings such as FastText and BERT word embeddings.

(2) Since during inference, the results can often be rather poor due to error propagation in the decoding process, we mainly propose two methods for solving that issue, once through a post-training calibration method called temperature scaling and through adapting a beam search algorithm.

(3) We introduce and publish a tool called *CATE* to demonstrate the work of our thesis. *CATE* allows users to parse a compositional tree from a phrase with a causal relation, while also being able to adjust the configuration of our model.

Kurzfassung

Requirements Engineering und Testen ist ein wichtiger Teil des Software-Entwicklungsprozesses. Jedoch kann das Erstellen von Testfällen von Softwareanforderungen sehr zeitintensiv und fehleranfällig sein. Um Entwicklungszeit und -kosten zu sparen kann automatische Extraktion von Ursachen und Effekten verwendet werden, um beim Erstellen der Testfälle zu unterstützen und zudem die Softwareanforderungen auf Fehlerredundanz, Abhängigkeiten oder Widersprüchen zu überprüfen. Diese Thesis leistet drei wesentliche Beiträge zur automatischen Extraktion von kausalen Relationen aus natürlich-sprachlichen Softwareanforderungen: (1) Experimente mit verschiedenen Einstellungen zur Optimierung von Tree Recursive Neural Networks (RNN) (2) Verbesserung der Inferenz von RNNs (3) Erstellung einer Demo um die Komposition von kausalen Relationen als Baum darzustellen.

(1) Wir führen verschiedene Experimente durch, um unser Model zu optimieren. Zunächst trainieren wir das Model mit zwei verschiedenen Datensätzen, dessen Bäume entweder links-verzweigend oder rechts-verzweigend erstellt wurden. Zudem ändern wir die Anzahl an Wortvektor-Dimensionen. Zuletzt trainieren wir das Model mit vortrainierten kontextunabhängigen und kontextabhängigen „Word Embeddings“, wie FastText und BERT.

(2) Während Inferenz kann Fehlerpropagation oftmals zu schlechten Ergebnissen führen. Hierfür betrachten wir zwei Methoden, um die Inferenzergebnisse zu verbessern. Zum einen die Post-Kalibrationsmethode „Temperature Scaling“ und durch Implementierung des „Beam Search“ Algorithmus in dem Decoder.

(3) Wir stellen ein Tool namens *CATE* vor und veröffentlichen dies, um die Ergebnisse unserer Thesis demonstrieren zu können. *CATE* erlaubt Nutzern Kompositionen von kausalen Relationen als Bäume darzustellen und dabei die Konfiguration des Modells zu verändern.

Contents

Introduction

1

1.1 Motivation

Software requirements are a crucial part of the software development process, since they ensure the quality and expectations of a product. These requirements are often divided into two categories – functional and non-functional software requirements. Functional requirements are product features that describe the software's functionality from the user's perspective as described in Fischbach et al. (2020) [14], they "usually specify a system from three perspectives: (1) the inputs to be processed by the system, (2) the expected system behavior once these inputs occur, and (3) the outputs that the system shall produce". In other words, functional requirements consists of a trigger or an action (cause) and an event (effect), which can be described as a causal relationship. From those requirements, developers derive test cases to verify if the software works as intended. Test generation is usually very time intensive and error-prone, taking anywhere from 40 - 70% of total development efforts [26]. With the help of causal relationship extraction, development costs could be significantly lowered and errors can be reduced. Fischbach et al. (2020) [14] outline two use cases for this matter:

Automatic Test Case Derivation is the process of deriving test cases from requirements. For example, if a requirement states "If the user is logged in and clicks on the analyze button, he will receive a summary of his activities.", the causes would be c_1 "the user is logged in" and c_2 "[the user] clicks on the analyze button" and the effect is e "[the user] will receive a summary of his activities". This test case could then be illustrated by a logical expression: $c_1 \wedge c_2 \implies e$. In this specific case, the requirement is defined as an implication $\implies$, however, equivalences $\iff$ can also be used to define a requirement.

Automatic Dependency Detection between Requirements: The second use case describes the detection of dependency between requirements. Two requirements could either contradict each other, require one another, be redundant, or one requirement could refine another. This will ensure, that test cases will be developed accordingly, and that problematic requirements will be revised before being implemented by the developers. Causality extraction can help automatic dependency detection by semantically comparing extracted causal relationship segments. For example, two causes or effects could be compared for semantic similarities or differences.

1.2 Problem Statement

There has been only very little research in causality extraction from natural language requirements. Existing method for general causality extraction fail to do so with high accuracy or fail to capture the logical meaning of a sentence, which are needed to build test cases. For instance, causes can have multiple sub-causes, and it is critical to represent those dependencies correctly since there are 2^n different combinations of n sub-causes and thus that many potential test cases. As a consequence, simply labeling cause and effect in a sentence will not be sufficient. Since the structure of natural language can often be described in a recursive manner, we will make use of a recursive neural network (RNN) as proposed by Socher et al. [42], that is able to parse a sentence as a constituency tree with labels for each composition.

1.3 Contributions

In this paper, we:

- experiment with various settings to fine-tune RNNs for automated causality extraction from natural language requirements. To the best of our knowledge, we are the first to use contextual word embeddings (BERT) for Recursive Neural Networks.
- explore ways to improve inference of RNNs, notably through the use of beam search and temperature scaling.
- introduce a tool, *CATE*, to parse the composition of a causal relation as a tree structure with the ability to change the configuration of our model.

1.4 Outline

In chapter 2, we will elaborate other approaches to the problem and discuss the similarities and differences between our paper and the related work. Chapter 3 will explain the fundamentals for this thesis. This chapter can be skipped if already familiar with the topic of this paper. After that, our solution approach will be explained in detail and the experiments that we conduct are presented. In chapter 5 we will evaluate the results of the experiments and explore which are improvements can be made. Chapter 6 introduces the web demo *CATE*. Chapter 7 concludes the research of this paper and discusses future work and research that can be made.

Related Work

2

In this chapter, an overview of related work will be provided. We will review work related to causality extraction and to natural language parsing separately.

2.1 Causality Extraction

A considerable amount of research has been published on the extraction of causality from natural language.

In a survey by Yang et al. (2021) [51], approaches for causality extraction have been categorized and evaluated. The three categories are *knowledge-based*, *machine learning-based* and *deep-learning based* approaches. Furthermore, Yang et al. mainly distinguish between explicit and implicit as well as intra-sentential (causal relationship in one sentence) and inter-sentential (causal relationships within multiple sentences) causality. Since in our paper, we will train a model on a single-sentence and explicit causality software requirements dataset, we will only review the explicit and intra-sentential approaches.

2.1.1 Datasets

Yang et al. describe the six most commonly used datasets for causality extraction models. The datasets and its features, disadvantages and limitations are listed in Table 2.1. SemEval-2007 Task 4 [15], SemEval-2010 Task 8 [20] and TACRED [53] lack an adequate amount of data. BioInfer [38] and ADE [19] [28] are not well-suited because they only cover the biomedical domain. The best option from the popular datasets would most likely be PDTB 2.0 [22] since it has the most causal sentences and is built in form of a treebank. But PDTB still originates from a general domain and lacks enough relevant fine-grained labels, which will lead to poor performance in RE tasks. For this reason, the RE Causality Treebank by Fischbach et al. [14] has been established. Further details of the dataset will be elaborated in chapter 4.

2.1.2 Knowledge-based Approaches

Knowledge-based approaches follow patterns or rules to extract causality. On one hand, this makes them more comprehensible from a human perspective, on the other hand, they are very labor-intensive and rather domain-specific with manual domain knowledge being required to find those patterns or rules. They use structure, lexico-semantic and syntactic analysis to find patterns

dataset	domain/type	size	advantages	limitations
SemEval-2007 Task 4 [15]	general/classification of semantic relations between nominals	140 training 80 test	strong reputation easily accessible	small amount of causal sentences
SemEval-2010 Task 8 [20]	general/multiway classification of semantic relations between pairs of nominals	1,003 training 328 test	strong reputation easily accessible	small amount of causal sentences imbalanced
PDTB 2.0 [22]	general/treebank	72,135 non-causal 9,190 causal examples	large corpus	stored in a complex way unmarked entities
TACRED [53]	general/annotated with person- and organization-oriented related type	269 causal examples		small amount of causal sentences
BioInfer [38]	biomedical/XML with entity markup	1,461 causal sentences		small amount of causal sentences
ADE [19] [28]	biomedical/relation about drugs and their effect	6,821 sentences with ADE relation	large corpus	
Causality Treebank [14]	software requirements/treebank with fine-grained labels	1,437 training 138 validation 141 test	well-suited for RE fine-grained labels treebank	relatively small amount of data

Table 2.1.: Datasets for causality extraction based on [51]

or make use of pre-defined keywords (e.g. *because*). In a paper by Khoo et al. [23], linguistic patterns for a wide range of interests were identified. Five ways to explicitly express causality were found, such as, causal links, causative verbs, resultative constructions, conditionals and causative adverbs and adjectives. After tokenizing a sentence, the tokens were pattern-matched with "slots" for cause and effects. In addition, these patterns were ranked from more specific to more general. This method achieved an average precision between 94-96% on a dataset of 1082 sentences from the Wall Street Journal, however, it has performed rather poorly on identifying causal relationships.

2.1.3 Machine Learning-Based Approaches

Machine Learning (ML)-based approaches can be less labor-intensive than knowledge-based approaches, but they still require manual efforts through feature engineering. Most ML-based approaches use some sort of third-party NLP toolkit, like StanfordCore, to engineer those features, which reduces manual labor. After that, a machine learning algorithm, like support vector machines, logistic regressions or maximum entropy is used. The paper by Blanco et al. (2008) [4], combines a pattern-based and ML-based approach. Using syntactic patterns, sentences that may encode causation are being identified. After that, machine learning is applied to classify between encoding and not encoding causation, resulting in an F1-Score of 91%.

2.1.4 Deep Learning-based Approaches

The latest addition to research on causality extraction has been the usage of Deep Learning (DL). Great results can be achieved through neural networks (NN) with a relatively low amount of labor, but one disadvantage is that compared to the other approaches, they are computationally expensive. In addition, they require a lot of data to work well, which only exists limitedly. The basic principle of using DL for causality extraction, is to map words and features to vectors and then train the model through, for example, convolutional neural networks (CNNs), recurrent neural networks or long short-term memory (LSTM) networks.

Kyriakakis et al. [27] leverage bidirectional Gated recurrent units (GRU), which are similar to LSTMs, with self-attention in combination with pre-trained models like ELMO [37] and BERT [10]. Similarly, Li et al. [29] have used a BiLSTM model with a multi-head self-attention layer and contextual word embeddings. Both papers have shown that models clearly benefit from using contextual word embeddings from pre-trained models, like ELMO, BERT or Flair [2].

There are two main differences of our paper, compared to other causality extraction approaches:

- We train and fine-tune our model explicitly for causality detection in software requirements.
- Our model is able to parse and label sentences in a tree-like form, resulting in much more fine-grained labeling.

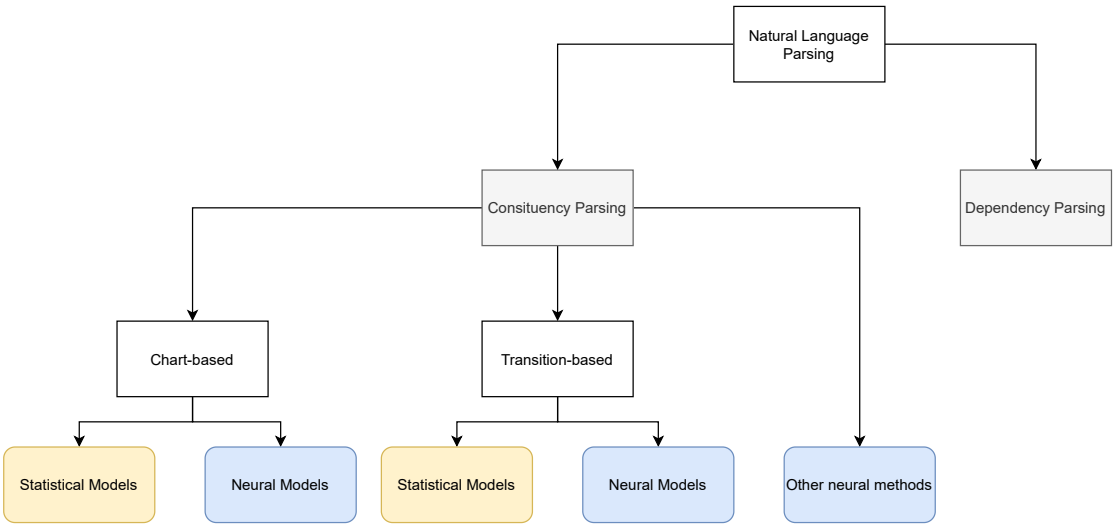

Figure 2.1.: Overview of different Natural Language Parsing approaches

2.2 Natural Language Parsing

Since we aim to parse the compositions in a tree-like structure from a natural phrase, our research is inherently related to the topic of Natural Language (NL) Parsing. NL Parsing can be broadly categorized into constituency and dependency parsing. Constituency trees illustrate the structure of a phrase by adhering to grammatical rules, while dependency trees or graphs represent the relation within a sentence between the word or segments of a sentence. Although dependency trees (See Figure 2.2 are better suited for extracting the direction of a causal relationship, constituency trees can depict nested causality better. In addition, constituency trees are easier to read and interpret by a human, than dependency trees. With some further work, it is also possible to derive dependency trees from constituency trees [9]. In a survey by Zhang [52], different approaches for constituent and dependency parsing are elaborated. Since we have chosen the constituent parsing approach, we will only present a selection of work related to that topic and not to dependency parsing. Zhang categorizes the approaches into *Chart-based* and *Transition-based*, with the two sub-categories *Statistical Models* and *Neural Models* (See Figure 2.1). Moreover, there is one category for other neural-based methods. Most models are benchmarked against the Penn Treebank (PTB) [31], a tree-based dataset containing around 7 million words, labeled with Part-Of-Speech (POS) tags. The metric that is used to compare the different models is the F1-Score.

2.2.1 Chart-based approaches

Chart-based approaches use dynamic programming to find an optimal tree. The most common algorithm used is the Cocke–Younger–Kasami (CYK) algorithm that finds an optimal solution in $\mathcal{O}(n^3)$ runtime, where n is the number of tokens [17]. The algorithm requires a set of

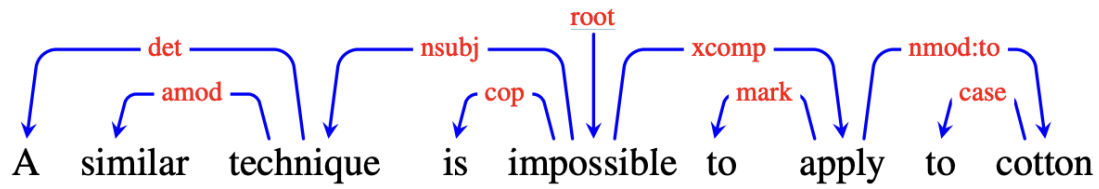

Figure 2.2.: Example for a dependency tree from [52]

Context-Free-Grammar (CFG) production rules that should be given in Chomsky normal form (CNF).

Statistical Models: Early efforts by Collins [7] and Charniak [6] extend Probabilistic CFG rules by adding a head word to each non-terminal node. The algorithm then maximizes the estimated probability of each PCFG rule that is used when making a parsing decision. These models are known as lexicalized models and have achieved an F1-Score of 88.2% and 89.5% respectively. Even better results have been achieved by McClosky et al. [33] using a slightly modified version of Charniak's parser which outputs the 50 best parses. In the second stage, using a Maximum Entropy model, a reranker classifies each parse based on features and then selects the best parse.

Neural Models: Socher et al.'s [40] first implementation of a Recursive Neural Network that defined scores over phrases attained an F1-Score of 90.4%. Most other approaches are not being trained in a recursive fashion. For instance, Stern et al. [44] train a bidirectional LSTM to label spans in a sentence, which are then decoded with CYK algorithm or Top-Down parsing approach. Kitaev and Klein [24] were able to develop a state-of-the-art model by using a similar approach, but by substituting the LSTM encoder with a self-attentive architecture. In addition, through pre-trained word embeddings, they could increase their F1-score from 93.5% to 95.1%.

2.2.2 Transition-based approaches

An advantage of transition-based parsers, is that they are often much faster than chart-based parsers, however, at least in the beginning they were regarded as less accurate.

Statistical Models: Traditional transition-based parsers use a shift-reduce system to construct a tree. The system consists out of a stack where partially constructed sub-trees are stored and a buffer for input words. A set of actions, such as $shift$ (pushes a word from the buffer into the stack) or *reduce* (combines two constituents and pushes the new constituent into the stack) is then applied to incrementally construct the final tree [39]. The parser is based on a greedy algorithm and beam-search can be used to reduce error propagation as done by Zhu et al. [54]. Discriminant classifiers are trained on manually engineered features (lexicalized head words, POS tags etc.) for transition action prediction. The model by Zhu et al. has achieved an F1-Score of 91.3% on the PTB.

Neural Models: Watanbe and Sumita [47] base their model on the one from Zhu et al. but the model is trained with a feed-forward neural networks. Cross and Huang [8] suggest the use of a

bidirectional LSTM encoder, which, at the time, achieved state-of-the-art results among greedy parsers. Other studies deviate from the traditional transition strategies and employ a top-down [11] or an in-order transition system [30]. Kitaev and Klein [25] reduce parsing to a tagging problem to maximize parallelism while still being able to achieve close to state-of-the-art results in linear time. Each node is tagged with one of four labels, that indicates whether it is a left-child or right-child for terminal and non-terminal nodes. The model is trained with pre-trained BERT word embeddings and a simple neural network to predict the tag, then two projection layers are applied to calculate a score for the actions. It is worth noting, that this model can only predict the sentence structure but not node-specific labels.

As can be seen, many constituency parsers have been developed throughout the years, relying on different approaches. In contrast to the above-mentioned constituency parse, we fine-tune our model not to parse the syntactic structure of a sentence, but to parse the composition of a causal relation for software requirements, which has not been researched in this setting before. The model that we will use is based on Socher et al.'s Recursive Neural Network (RNN) for sentiment analysis [42]. Evidently, using (contextual) word embeddings can have a great positive impact on the results, both for NL parsing and causality extraction. For this reason, the usage of pre-trained word embeddings in combination with an RNN will be explored, as well as different parsing decoder technologies like beam search.

Fundamentals

3

In this chapter, we will explain some fundamental knowledge and terminology, in order to understand this paper better with limited prior expertise in this field.

3.1 Causality

Causality in natural language describes the relationship between a cause and an effect. These relationships can occur in many forms, which we will briefly define in the following, according to Blanco et al. (2008) [4].

3.1.1 Marked vs. Unmarked Causality

Marked causality always has a distinguished, linguistic unit which indicates the causality. Consider the sentences:

> "The user received an email *because* he was registered."
>
> "*If* logged in, the user shall be able to see his activity."
>
> "*In order to* stop a process, a task has to be selected first."

In the first sentence, *because* indicates the causal relationship and thus the sentence is marked. The other sentences are also marked by other linguistic units like *If* or *In order to*. Contrary, an unmarked phrase would be:

> "Don't smoke. It can kill you."

3.1.2 Ambiguity of Causality

Ambiguity in causal relationships can be defined by whether the cue phrase also has a meaning which does not indicate causality. The linguistic unit *because* always expresses causality, however, a word such as *when* does not, since it can also just refer to a time description. The following sentence, illustrates an example with an ambiguous cue phrase (*since*) that does not refer to causality but only indicates a time. In contrast, in the next sentence, *since* does indicate a causal relationship.

> "*Since* 2012 the S&P 500 has tripled in value."
>
> "*Since* logging metrics are important, they shall be displayed on a separate page."

3.1.3 Explicit vs. Implicit Causality

In an explicit phrase, cause and effect are always clearly stated, such as in the following phrase.

> "Because the user stopped the task, the process was terminated."

In the sentence,

> "The user stopped the task."

the effect, "the process was terminated", is only implicitly mentioned and requires some background knowledge to be induced by the reader.

In this paper, we will only be focusing on and training our model for *explicit* and *marked* causality, since functional software requirements most frequently occur in that form.

3.2 Requirements Engineering

> "Requirements engineering is the process of conforming engineering designs to a set of core software requirements. This is critically important for creating accurate results in software engineering." [48]

Requirements Engineering (RE) is the first step of the software engineering lifecycle [43] and consists out of multiple subfields and tasks, including requirements elicitation, analysis, specification, validation and management [32] [48]. A common reason for failure in software development is a bad RE process, showcasing the importance of good RE for successful software development. During RE, requirements and restrictions will be specified which are validated for various criteria such as consistency (conflicting requirements), completeness (do the requirements cover the whole system?) and detectability (are the requirements well documented so that they can be detected?)[32]. Software requirements can either be functional or non-functional requirements. Functional requirements represent the interaction between a user and a system, which are often conveyed through causal relationships. Non-functional requirements are features, that do not describe the interaction of user and system, but rather refer to aspects like the performance of a system. An example for a functional requirement is "If the users submits his form, the application shall be processed by the system." and a non-functional requirement would be "The system shall give a response in under 10 seconds". After the requirements are specified and validated in the RE process, and later implemented, they are tested for correct functionality to assure the quality of the software.

3.3 Deep Learning

Deep Learning (DL) is a subset of Machine Learning and Artificial Intelligence (AI) and is used to learn specific tasks that otherwise can be solved by a human. DL usually utilizes (Artificial or Deep) Neural Networks (NN). Similarly to a human or animal brain, neural networks consists out

of artificial nodes, referred to as neurons. These neurons are coordinated in multiple activation layers, some of them are "hidden" between the input and output layer, which is where the term *deep* comes from. Each neuron represents a number, and some layers are connected with "weights" (matrices) that are updated as the NN is being trained and learns.

3.4 Natural Language Processing

Natural Language Processing (NLP) is a sub-field of Linguistics and Artificial Intelligence (AI), which enables machines to understand, interpret, generate or manipulate natural language. NLP has been first used in the 1950s and has risen out of linguistics field, with the rise of power in computers to solve task such as language translation. [1]

A major breakthrough in NLP was the shift from statistical methods, which relied mainly on tasks such as feature engineering, to DL and NNs with which semantic meaning can be better captured.

3.5 Recursive Neural Networks

Recursive Neural Networks (RNN; not to be confused with Recurrent Neural Networks which often uses the same abbreviation) have been first introduced by Goller and Küchler in 1996 [16] but have only become more popular in recent years through the research of Socher et al. although they are still not regularly used for real-word or production tasks. RNNs are used for inference of structured data by recursively applying a neural network on an input. An RNN uses a composition function to merge two nodes into one. This composition function is further extended by Socher et al. to get two sub-versions of the conventional RNN; Matrix-Vector RNN (MV-RNN) and Recursive Neural Tensor Network (RNTN). However, in this thesis, we will be using a traditional RNN as our base model. The exact implementation of the RNN used in this paper will be elaborated in chapter 4. Previous tasks of RNNs include parsing of natural scenes and natural language [41] or for sentiment prediction [42].

3.6 BERT

BERT, short for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers, is a state-of-the-art language model developed by Google [10]. The model is heavily based on the concept of Transformers [46]. The Transformer is a transduction model that converts input sequences into output sequences, solely relying on its self-attention algorithm. There is a transformer block at each of BERT's 12 or 24 layers, depending on the model size (base or large). BERT first tokenizes a sentence using WordPiece embeddings, [50] which has a 30 000 token vocabulary. Each token is then constructed of a token, segment and position embedding. BERT is trained on two unsupervised prediction tasks, which we will explain briefly:

Masked Language Model (MLM): Since standard condition models can only be trained left-to-right or right-to-left, BERT uses a Masked Language Model to train its deep bidirectional

representation. To do so, 15% of a sequence's token are chosen at random, which are then being masked with a [MASK] token 80% of all times. In 10% it is replaced with another word and in the other 10% the original word is kept to avoid a mismatch between pre-training and fine-tuning. Using the cross-entropy-loss, the model is then trained on predicting the masked tokens.

Next Sentence Prediction (NSP): In order to understand sentence relationship, the model is also trained on a Next Sentence Prediction task. Two sentences are chosen at random, where sentence B follows sentence A in 50% of all times. The model then tries to predict whether sentence B follows sentence A or if it does not.

BERT can be further fine-tuned for various NLP tasks with just one additional layer, however, since we will not be fine-tuning BERT embeddings, we omit further detailed explanations. All in all, it can be said that BERT is a desirable addition to our model due to its bidirectional nature and state-of-the-art performance.

Extraction of Causal Relations from Text 4

4.1 Tools

4.1.1 Python

Python is a high-level, general-purpose programming language. It is chosen widely for data science and deep learning projects because of its readability and simplicity, which allows developers to focus on the research process. Furthermore, Python offers a wide range of frameworks and libraries for high-performance scientific computing, machine learning, and data visualization, thus making it the perfect tool for our paper. [49]

4.1.2 PyTorch

PyTorch [35] is an open-source machine learning framework for Python which was primarily developed by Facebook's AI Research lab and is being used by an extensive scope of real-word AI applications like Tesla Autopilot. PyTorch provides many modules which can significantly accelerate the development process by completely obviating the need of writing complex back-propagation algorithms from scratch or implementing functions, like ReLu, Softmax or the loss function, while still allowing maximum flexibility. In addition, PyTorch integrates well with other tools like TensorBoard, TensorFlow's visualization and logging toolkit, with which metrics like loss or accuracy can be easily represented in a graph. Unfortunately, PyTorch lacks an opinionated code structure which is why we will also be using PyTorch Lightning [12], a library that can help organize PyTorch code better, removes a lot of boilerplate code, is less error-prone, and makes scaling hardware easier.

4.2 Dataset

Tree Recursive Neural Networks require strongly structured and labeled binary trees as an input for training. For this purpose, a gold standard corpus has been established by Fischbach et al. (2020) [14]. The data has been sourced from 463 different requirement documents, manually filtered for causal relationships, and annotated with 28 labels. The labels and the meaning of each label are listed and explained in the appendix A.1. At the time of writing, the dataset has a total of 1716 sentences. These sentences were split into train, test and validation splits as shown in Table 4.1:

Split	Sentences
Train	1437
Validation	138
Test	141

Table 4.1.: Train, Validation and Test Split

To validate, that the sentences are equally distributed in the dataset splits, we can examine the distributions of the labels for each split in Figure 4.1. As can be seen, the distributions among all three splits are similar and therefore are suitable to use for our model. The most frequent labels is, as expected, the *Word* label, followed by the *Condition*, *Variable* and *Statement* label. Some labels are very rare, like *Separated Or* or *Separated Effect* and do not even occur in the validation and test dataset, and thus we can expect worse results on those labels as more data is needed. However, since these labels are not particularly important as they are sub-labels, the dataset is still perfectly acceptable for our use case.

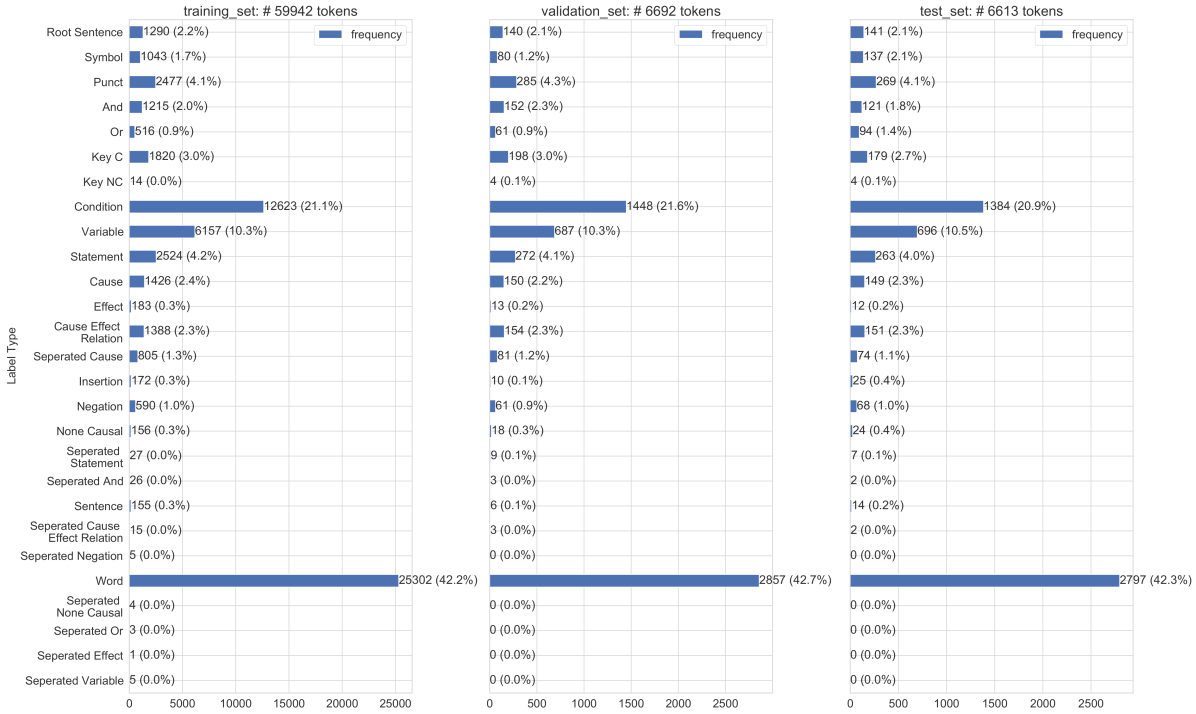

Figure 4.1.: Distribution of Labels

Figure 4.2 illustrates the mean and standard deviation for the number of tokens per label. For example, the "Statement" segment in Figure 4.4 consists out of 5 and the "Root Sentence" segment of 11 tokens. Labels with fewer tokens, like "Word" or "Punct", are more trivial to predict than labels with sequences of more tokens. However, longer segments do benefit from having more

contextual information.

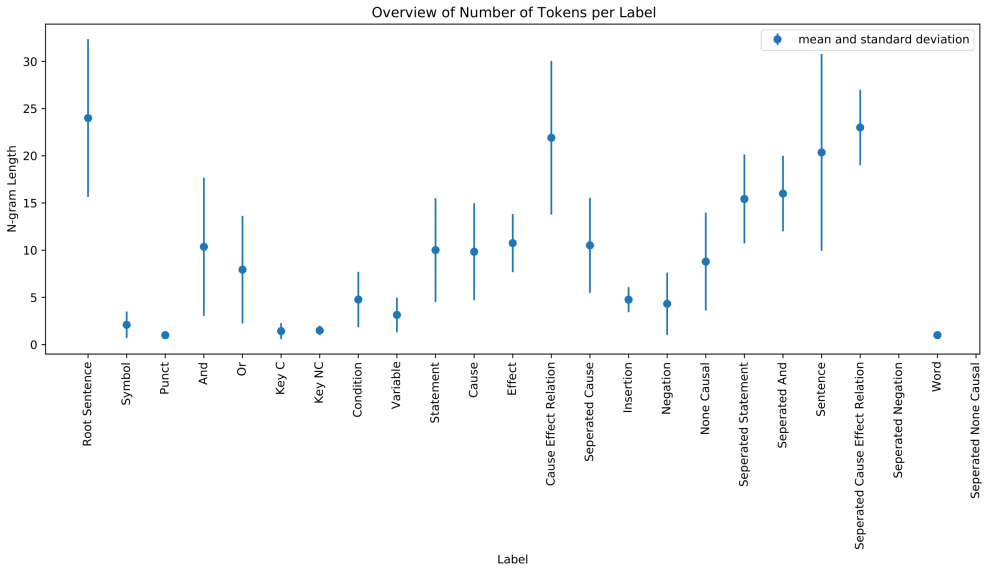

Figure 4.2.: Tokens per Label

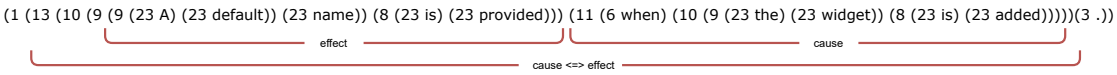

Figure 4.3.: Annotation scheme of the dataset. The string can then be parsed as a tree.

An RNN requires binary trees as input for training. These trees can be built using two different branching methods: left-branching and right-branching. Let us consider the three tokens “A default name”, which constitute a variable segment (see Figure 4.4). An RNN can only join adjacent token pairs in each recursion, resulting in two merge options for these tokens. Either “A” and “default” are merged first (left-branching, see Figure 4.4) and then connected with “name” or “default” is first connected with “name‘” and then joined with “To” (right-branching, see Figure 4.5). Two different datasets were built for both approaches.

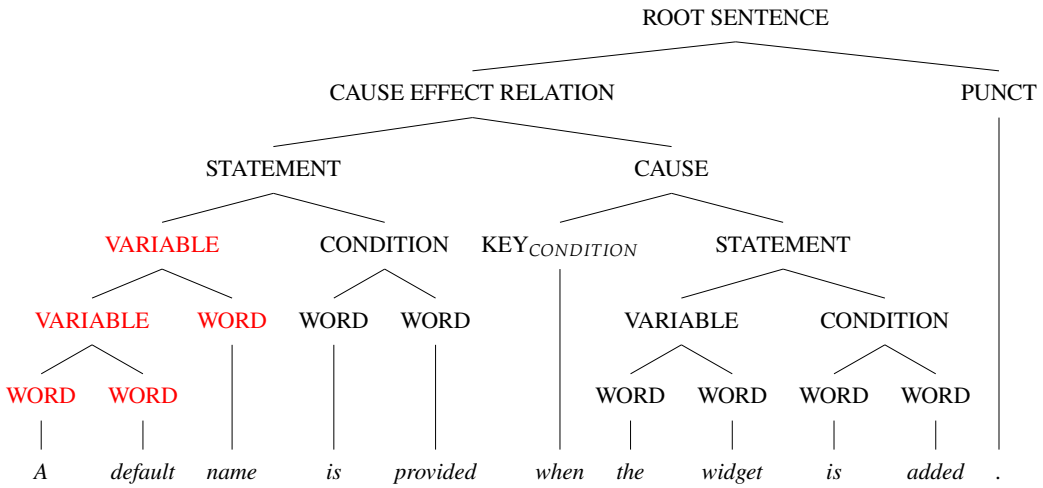

Figure 4.4.: Sample of the left-branching dataset

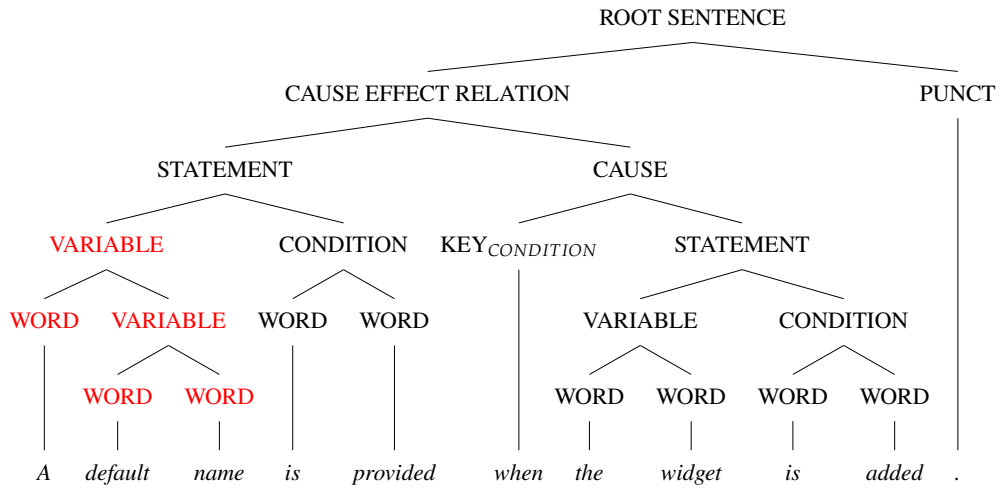

Figure 4.5.: Sample of the right-branching dataset

4.3 Model Overview

In this chapter, we will elaborate how the model is structured and how it functions. As mentioned before, an RNN recursively applies a neural network to an input at each node in a tree-like structure.

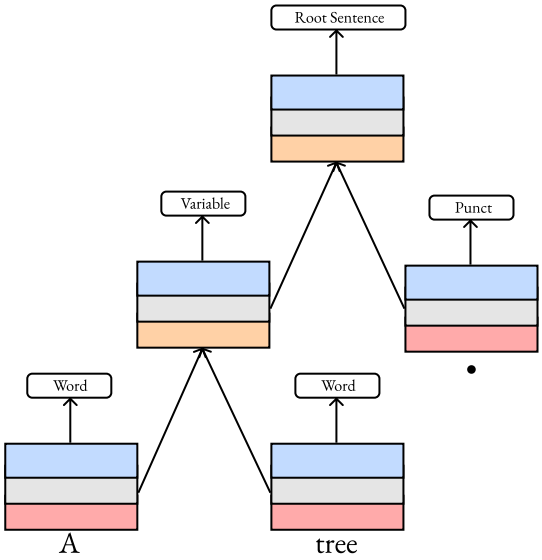

Figure 4.6.: Overview of the Tree Structure

4.3.1 Neural Network

The neural network has three different layers, whereas the first layer is different depending on whether the node is a terminal (i.e. leaf) or non-terminal (i.e. inner) node.

The first layer of the internal nodes' neural network is the **Embeddings Layer**. The input or word embedding is a vector from the word embeddings matrix $L \in R^{d \times |V|}$ where $|V|$ is the size of the vocabulary in the dataset and d is the number of dimensions of each word vector. The vectors are initialized from the normal distribution: $\mathcal{N}(0,1)$.

The **Composition Layer** is the first layer of the internal nodes NN. First, the output of two child nodes are being concatenated, which results in the vector x. For each neuron z_j, where $j \leq d$, with the input x_i, where $i \leq 2d$, we compute:

$$f(x_i) = z_i = W_{ji} x_i \tag{4.1}$$

The **Activation Layer** applies an activation function to our input. Although, Socher et al. describe in their paper that they use the $tanh$ function, we use the nowadays more regularly used *Rectified Linear Activation Function* (*ReLU*) which returns the output of $max(0, z_j)$ where z_j is the input of the function.

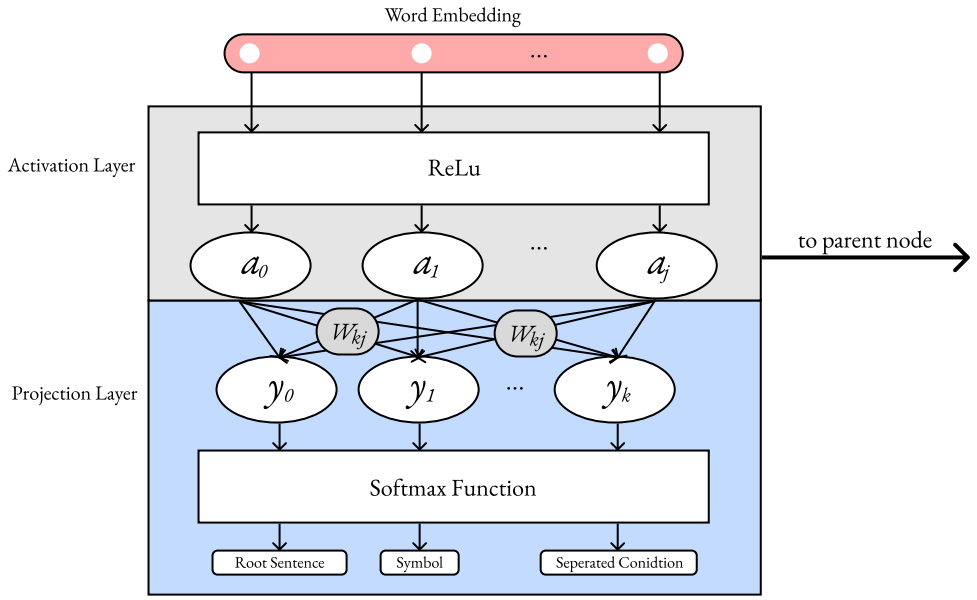

Figure 4.7.: Neural Network block of a terminal node

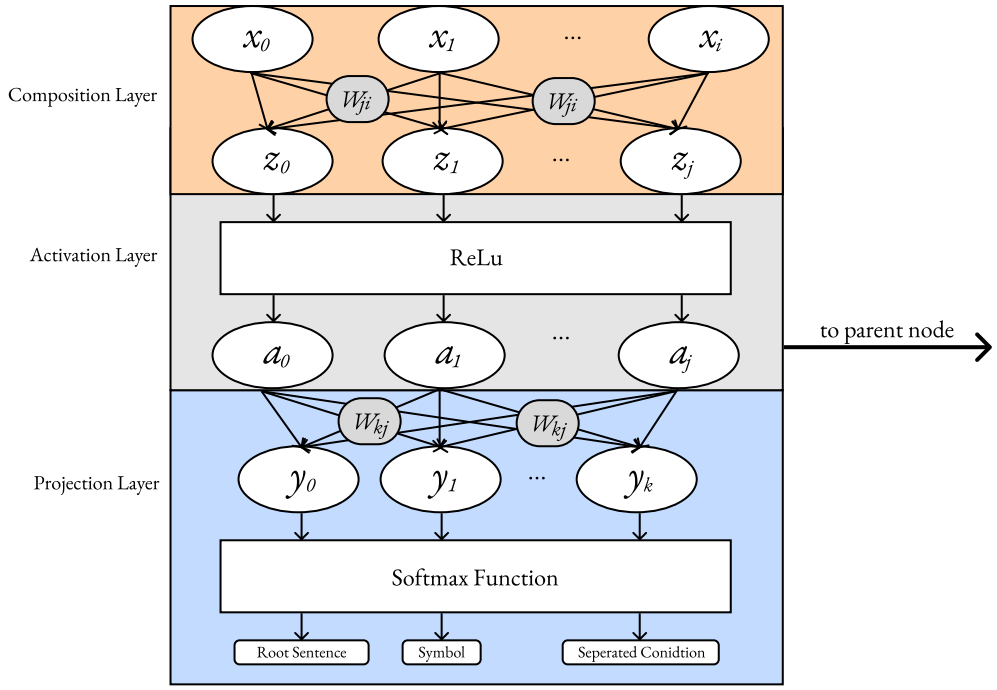

Figure 4.8.: Neural Network block of a non-terminal node

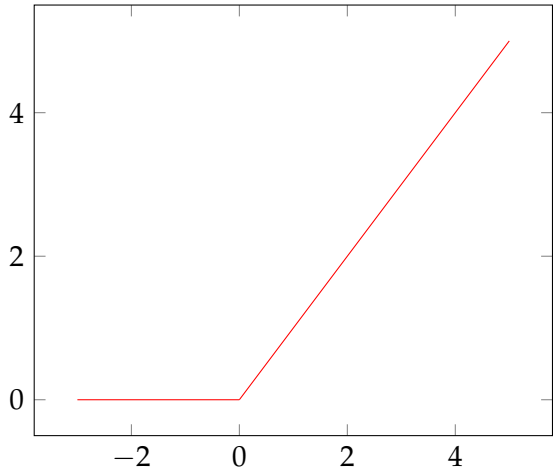

Figure 4.9.: Rectified Linear Activation Function

Using ReLU has a few compelling advantages over traditional activation functions like *sigmoid* or *tanh*. (1) A *max* function is more trivial to compute than exponential calculations. (2) ReLU can output true zero values, which simplifies the model and accelerates learning, especially for representational learning like classification tasks. (3) The model is easier to optimize since, in contrast to *sigmoid* and *tanh*, it is a linear function.[5]

The output of the Activation Layer is passed on to the *Projection Layer* and to the next parent node, where it is merged with its sibling node in the *Composition Layer*.

The **Projection Layer** is used to get a classification output for each node. First, the output a of the activation layer is multiplied by the label classification matrix $W_l^{28\times d}$ for 28 different labels (or a_j by W_{kj} on a neuron basis). With the output, y_k, where $k \leq 28$, we can calculate the posterior probability of each class by feeding it to the softmax function in 4.2.

$$\text{Softmax}(y_k) = o_k = \frac{\exp(y_k)}{\sum_{m=1}^{28} \exp(y_m)} \tag{4.2}$$

4.3.2 Model with pre-trained Word Embeddings

Word embeddings represent the meaning of a word in neural networks, most commonly in the form of a vector. In Socher's RNN, these word embeddings are being trained from the words of the dataset [42] from the ground up in the *Embeddings Layer* after being initialized randomly. We will refer to those embeddings as random word embeddings. However, random word embeddings have two major disadvantages.

First, the word embeddings are limited to the corpus of the dataset, meaning that every word which is not included in the dataset has to be encoded as an "unknown word" token (UNK) which leads to loss of important information and potentially to mislabeling or incorrect parsing of a

sentence. Secondly, with random word embeddings, any kind of contextual information is lost. For example, the word "ruler" has two entirely different meaning depending on its context, one being an object which is used for measuring and the other being a person who governs a group of other people. The meaning depends entirely on the context of the sentence, but both will receive the same word vector.

To address these issues, we can leverage pre-trained word embeddings, which are word-embeddings that have been trained in advance, usually on a larger corpus, and can then be used in the model, either fixed in the current state or further fine-tuned. Instead of the *input layer*, the NN will just receive a vector from the pre-trained word embeddings as input. Two common word embeddings are FastText [34] (1M word vectors) and GloVe embeddings [36] (2.2M word vectors). Though, because of the large quantity of words, they can solve the first problem, the second problem still remains. A solution for the second problem is to use contextual pre-trained word embeddings like BERT [10] embeddings. Using BERT will change the architecture of our RNN, as illustrated in Figure 4.10. Before training and inference, a sentence will first be tokenized by the BERT tokenizer, which may split a few words in multiple tokens and consequently change the structure of the tree. Afterward, we apply the BERT model to our sentence, which will then output the 768-dimensional BERT_{base} word embedding vectors that are passed to the NNs.

4.3.3 Loss Function

As a loss function, we will use the **Cross-Entropy-Loss** which is commonly used in multi-classification models. It is defined as followed:

$$\mathrm{E}(o_k) = -\sum_c t_c \log o_k \tag{4.3}$$

where t_k is the one-hot encoded gold probability. o_k simply is the output of the softmax function. By replacing o_k with the softmax function and rearranging the equation, we get the following:

$$E(y_k) = -\sum_c t_c (y_c - \log \sum_{c'} e^{y_{c'}}) \tag{4.4}$$

4.3.4 Backpropagation

Backpropagation is a method used by feed-forward neural networks to calculate the weight changes, given an error function, which in our case is the Cross-Entropy-Loss defined above. The weight updates can be calculated by a simple **gradient descent** algorithm like in 4.5 and 4.6, where μ is the learning rate of the model.

$$\hat{W} = W - \Delta W \tag{4.5}$$

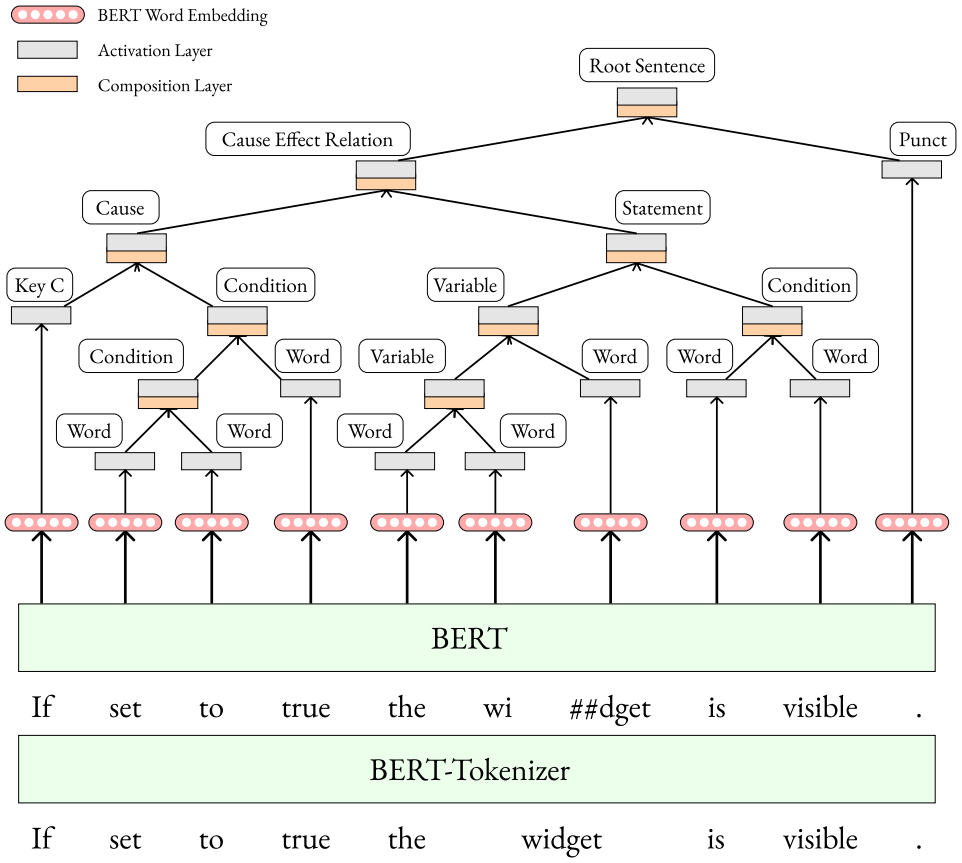

Figure 4.10.: RNN architecture with BERT embeddings (Projection layer left out for simplicity)

$$\Delta W = \mu \frac{\partial E}{\partial W} \tag{4.6}$$

There are two weights that need to be trained in our model, as well as the word embeddings matrix for the models that do not use pre-trained embeddings. The first weight is W_{kj} which is used for classification. We will use the chain rule to calculate the derivate of our error function in respect to this weight.

$$\frac{\partial E}{\partial W_{kj}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial y_k}\frac{\partial y_k}{\partial W_{kj}} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial W_{kj}} \tag{4.7}$$

$$\begin{aligned}
\frac{\partial E}{\partial y_k} &= -\sum_c t_c(\mathbb{1}_{c=k} - \frac{1}{\sum_{c'} e^{y_{c'}}} e^{y_k}) \\
&= -\sum_c t_c(\mathbb{1}_{c=k} - o_c) \\
&= \sum_c t_c o_k - \sum_c t_c \mathbb{1}_{c=k} \\
&= o_k \sum_c t_c - t_k \\
&= o_k - t_k = \delta_k^L
\end{aligned} \tag{4.8}$$

4.9 is the identity function, which is either 0 or 1.

$$\mathbb{1}_{c=k} = \begin{cases} 1, & \text{if } c = k \\ 0, & \text{otherwise.} \end{cases} \tag{4.9}$$

In the next step, we derive y_k in respect to W_{kj}:

$$\frac{\partial y_k}{\partial W_{kj}} = a_j \tag{4.10}$$

Thus, we get the weight change:

$$\Delta W_{kj} = \mu(o_k - t_k)a_j \tag{4.11}$$

To update the second weight, W_{ji}, we follow similar steps as above by solving the equation in 4.14

$$\begin{aligned}
\frac{\partial E}{\partial W_{ji}} &= \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial y_k}\frac{\partial y_k}{\partial a_j}\frac{\partial a_j}{\partial z_i}\frac{\partial z_i}{\partial W_{ji}} \\
&= \delta_k^L \; W_{kj} \; ReLu'(z_i) \begin{bmatrix} a_j^{left} \\ a_j^{right} \end{bmatrix}
\end{aligned} \tag{4.12}$$

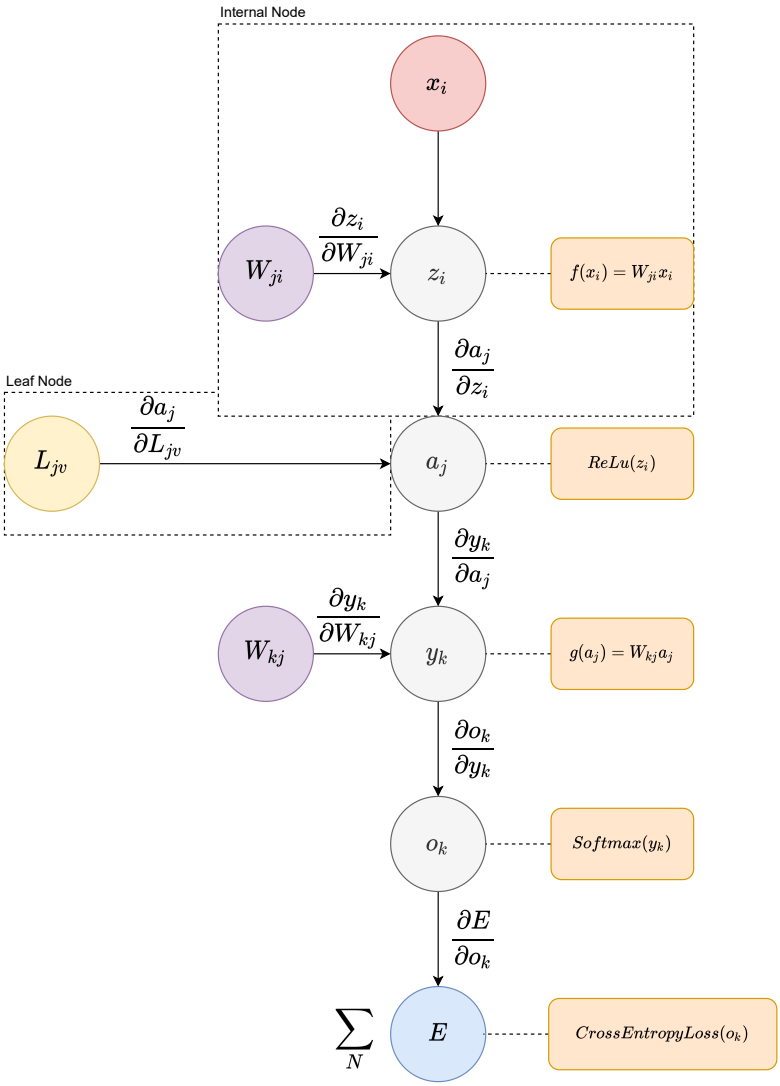

Figure 4.11.: Computational Graph for Backpropagation

where a_j^{left} and b_j^{right} are the recursively computed outputs of the child nodes and the derivative of the $ReLu$ function is:

$$ReLu'(z_i) = \begin{cases} 0 & z_i < 0 \\ 1 & z_i > 0 \end{cases} \tag{4.13}$$

Analogically, the word embeddings matrix update is calculated as followed:

$$\frac{\partial E}{\partial L_{jv}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial y_k} \frac{\partial y_k}{\partial a_j} \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial L_{jv}} \tag{4.14}$$

where v is the index of the word in the embeddings matrix. This algorithm is applied at each node in a top-down fashion after a sentence was fed-forward to the RNN.

Stochastic Gradient Descent: Simple gradient descent optimization is slow since it has to make a lot of calculations and thus is rarely used in complex models. To improve training time and performance, other optimization algorithms, such as Stochastic Gradient Descent (SGD) optimization, are used in practice. In contrast to normal gradient descent, SGD stochastically estimates the gradient by only using a random subset from the dataset. Furthermore, SGD can be used in combination with the momentum method. The momentum method can accelerate gradient descent by calculating the next weight update while considering the gradient of the last update. The specific implementation of the momentum technique used is based on the paper of Sutskever et al. (2013) [45]

4.4 Experiments Setup

4.4.1 Training Environment

All models were trained on an Ubuntu server with a 10-core Intel Xeon Skylake Processor, clocked at 2.4 GHz and 46.25 GB of usable memory.

4.4.2 Metrics

There are five metrics that we will monitor and evaluate. During training and validation, the **loss** will be monitored closely, of which the calculation has been explained in the prior section. The objective is to minimize training and validation loss. However, a declining training loss and growing validation loss could be an indicator for over-fitting, which is why more attention will be paid to the validation loss.

Another important metric is **accuracy**. Accuracy can be simply calculated by this formula:

$$\text{accuracy} = \frac{\text{number of correct predictions}}{\text{number of total predictions}} \tag{4.15}$$

. It can be a good indicator for training, as well as validation and testing.

One issue with accuracy is that it can be easily inflated if, for instance, each prediction is labeled as *Word*, we will solely achieve an accuracy of over 42% since this is the amount of the *Word* label out of all tokens (see Figure 4.1). To solve this problem, it is important to validate the performance of our model on a per-class basis with the three metrics *precision, recall* and the *F1-Score*.

For each class, we define the following:

$$tp : \text{number of true positives} \tag{4.16}$$

$$fp : \text{number of false positives} \tag{4.17}$$

$$fn : \text{number of false negatives} \tag{4.18}$$

These can then be used to calculate precision and recall.

$$\text{precision} := \frac{tp}{tp + fp} \tag{4.19}$$

$$\text{recall} := \frac{tp}{tp + fn} \tag{4.20}$$

Intuitively, **precision** is the ability of our model not to falsely label a positive sample, while **recall** is the model's ability to find all positive samples. The meaning of both those metrics can be captured as the weighted average by the **F1-Score**.

$$F_1 := 2 \times \frac{precision \times recall}{precision + recall} \tag{4.21}$$

These metrics can be averaged across all classes to get an impression of the whole model's performance at a glance. There are three commonly used averaging methods: the *micro-average, macro-average* and *weighted-average* method.

Since the **micro-average** method sums up all tp, fp and fn respectively, the metrics will be identical to the accuracy in multi-classification settings. The **macro-average** on the other hand is defined as the unweighted mean for each metric of all classes and could be misleading if the classes are unbalanced like in our cases.

The best average method for this model is the **weighted-average** method, which is the weighted mean by the true number of instances for each label and as a consequence will give us a better impression of precision, recall, and F_1 for all classes.

4.4.3 Experiments

During our experiments, we will focus on changing two parameters. The first one is the dataset on which we will train our model. As mentioned before, the trees can either be left-branching or right-branching, hence there are two options for the dataset. The second variable is the word embeddings type, which can either be d-dimensional random word embeddings, GloVe or FastText embeddings (300 dimensions), or BERT embeddings (768 dimensions). The random word embeddings will be trained with $d \in \{30, 60, 300\}$.

In total, there will be 12 different experiments, as seen in Table 5.2.

dataset	embeddings	dimensions
left branching	random	30
left branching	random	60
left branching	random	300
left branching	GloVe	300
left branching	FastText	300
left branching	BERT	768
right branching	random	30
right branching	random	60
right branching	random	300
right branching	GloVe	300
right branching	FastText	300
right branching	BERT	768

Table 4.2.

Each experiment will be trained for a maximum of 50 epochs and training will be stopped early if the validation accuracy does not increase after 10 epochs. After training is finished, the checkpoint model with the lowest validation loss will be tested for all above-mentioned metrics. In initial experiments, we have already determined a suitable learning rate of 0.01 and momentum of 0.9.

Evaluation

5

In this chapter, we will evaluate the results of the experiments and demonstrate improvements that can be made for inference.

5.1 Left vs Right-Branching Dataset

First and foremost, we will assess if the kind of dataset used has any impact on the outcome of the model. To observe this, we will group the experiments by left-branching and right-branching dataset and take the mean of the metrics. As seen in the validation accuracy graph in Figure 5.1 and the validation loss graph in Figure 5.2, there is a clear benefit of using the left-branching dataset. On average, the model trained on the left-branching datasets has performed 4.5% better on the test dataset. Furthermore, without any exceptions, each experiment has seen better results on the left-branching dataset than on their counterpart. For this reason, in the next section, we will only evaluate the 6 experiments on the better performing dataset and not interpreting any results from the right-branching dataset.

Figure 5.3 shows that on the right-branching dataset, the accuracy is declining the more tokens a tree has, while this is not the case (or to a smaller degree) for the model trained on the left-branching dataset. Additionally, we have noticed that there is a big drop in performance for the condition label (see Table 5.1). In many cases, condition segments have to be branched as explained in chapter 4.2 and the more tokens a tree has, the more likely it is that it also has long segments, thus resulting in lower accuracy for longer sentences. The reason for that can be explained as followed. Models using left-branching data observe words and context starting from the beginning (left side) of a segment, while the context of the last words is only perceived at the end. We assume that in the English language, words at the beginning of a segment, have a higher contextual importance than the ones at the end. Let us consider the statement “English is selected as default”, where “is selected as default” is a condition segment. A right-branching model would start concatenating the words “as” and “default”. These words alone do not suggest that the segment is a condition, however, when starting from the left, “is selected” would rather give us the perception of a condition. Since with BERT, the word embeddings are already contextual, the decline in performance is lower than with other word embeddings. This also supports our hypothesis, that our model benefits from contextual word embeddings.

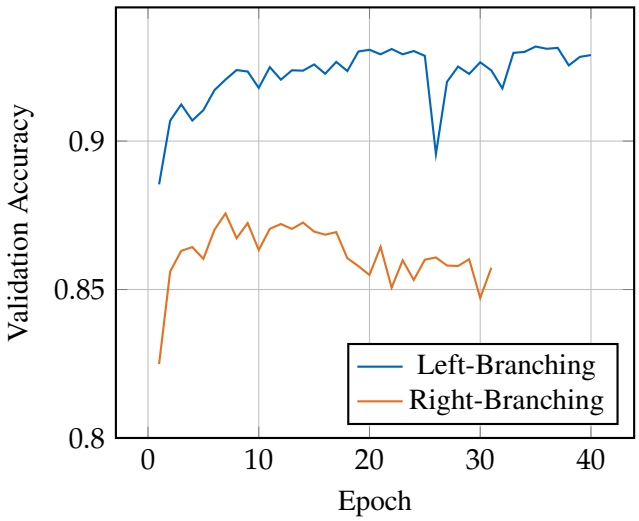

Figure 5.1.: Average Left-Branching vs. Right-Branching Validation Accuracy.

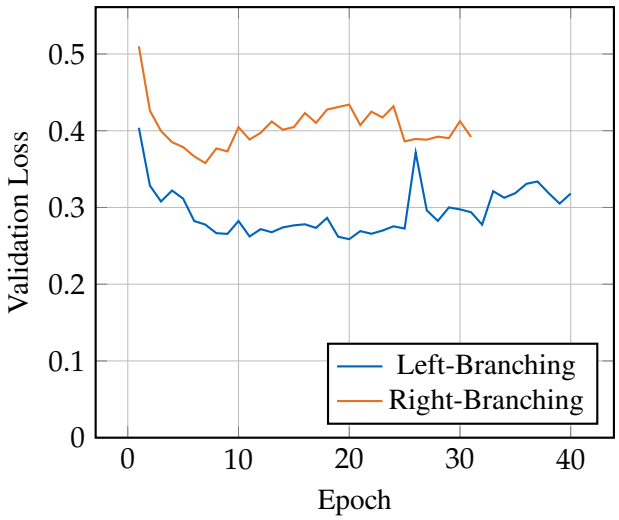

Figure 5.2.: Average Left-Branching vs. Right-Branching Validation Loss.

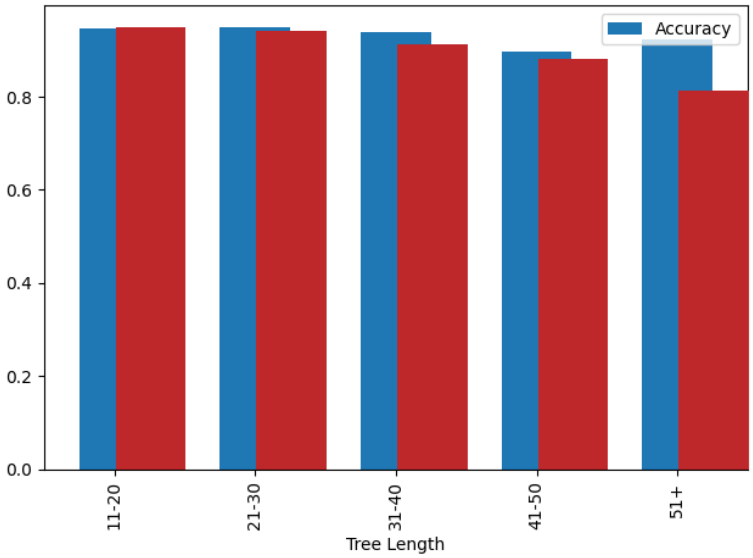

Figure 5.3.: Average Accuracy per Token per Tree for FastText embeddings. Left-branching accuracy in blue and right-branching accuracy in red.

	F1-Score left-branching	F1-Score right-branching	Difference
random (30)	**88%**	76%	12%
random (60)	**90%**	76%	14%
random (300)	**90%**	77%	13%
GloVe (300)	**91%**	71%	20%
FastText (300)	**92%**	78%	14%
BERT (768)	**94%**	89%	5%

Table 5.1.: Left-branching vs Right-branching F1-Scores for condition label (rounded)

5.2 Dimensions

In this step, the dimensions hyper-parameter will be compared on random word embeddings. While the 300-dimensional model F1-Score was 2.43 percentage points and 0.3 percentage points better than the 30 and 60-dimensional models respectively, the improved results come at the cost of training time. Training time of both the 30 and 60-dimensional models has been exceptionally fast, while the 300-dimensional model took more than 12 times as long as the smaller model.

	random(30)	random(60)	random(300)	GloVe(300)	FastText(300)	BERT(768)
training time	**5m 54s**	10m 51s	1h 11m 51s	1h 56m 18s	1h 8m 28s	1h 33ms 54s
avg per epoch	**19.67s**	21s	349.5s	174.45s	164.32s	225.36s
total epochs	31	**17**	**17**	20	24	**17**
best epoch	17	17	**4**	13	11	6
precision	89.84%	91.73%	92%	93%	92.02%	**94.01%**
recall	90.38%	92.24%	92.67%	93%	92.59%	**94.24%**
F1	89.78%	91.91 %	92.21%	93%	92.25%	**94.07%**

Table 5.2.: Left-branching detailed training results with weighted average metrics.

5.3 Word Embeddings

Since the 300-dimensional random word embeddings experiments has performed best from all three random word embeddings models, we will compare that model to the other word embedding models.

Interestingly, neither the GloVe, nor the FastText embeddings performed significantly better or worse, than the random embeddings, yet one benefit of using those pre-trained word embeddings, is the reduction of training time per epoch since they do not have to be further fine-tuned and thus leads in a model with fewer parameters. The average time per epoch was cut in half but, at least in our experiments, it took more epochs for the model to converge. In addition, we can expect better results in inference for sentences with words that are not included in the corpus. Fundamental improvements can be observed by the usage of BERT embeddings. The F1-Score has risen from 92.21% to 94.07% and training time per epoch is still lower than the random word embeddings model. The small difference in performance between the 60 and 300-dimensional model, indicates that the BERT embeddings model does not simply outperform the other models due to the number of parameters. Weighted-average recall, precision and F1-Score has also improved compared to all other experiments. This confirms our thesis that our model can benefit from the use of contextual word embeddings.

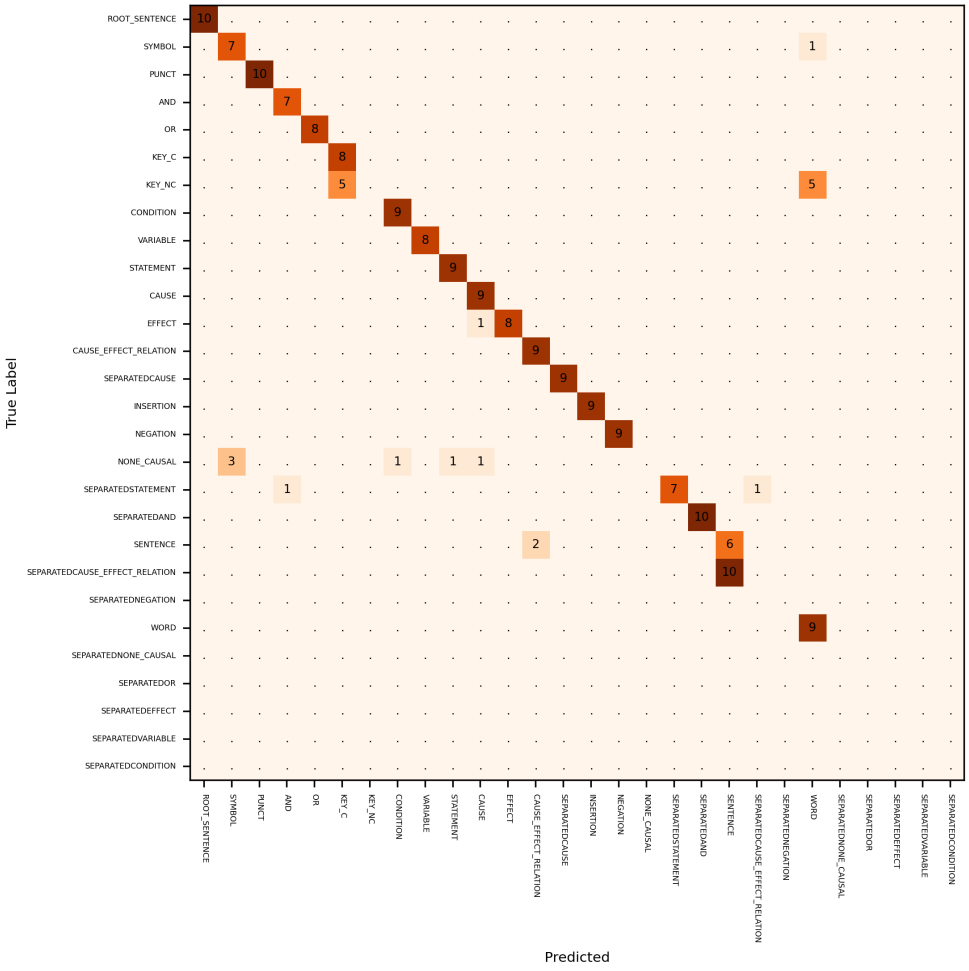

Figure 5.4.: BERT Confusion Matrix (left-branching dataset)

To further examine the results, we can take a look at the confusion matrices. A confusion matrix is a table where, for each label, the predicted results are compared to the true values. In a perfect scenario, we would see a diagonal line from the upper left to the lower right without any interruptions. The color of each block indicates how strong the correlation is between a predicted and true label. Comparing the confusion matrices on the test dataset of the four experiments, the diagonal line of the BERT word embeddings model is the strongest. For most predicted labels, there is a strong correlation with the corresponding true labels, however, for some that is not the case. In the following, we will evaluate each label.

Root Sentence: Predicting the Root Sentence label is a relatively trivial task for our model, as it always follows a clear structure (Rest of the sentence + punctuation).

Symbol: Although it would seem like predicting the Symbol label is easy, the model could only do so with a precision of 67% and recall of 72%. Since a simple Regular Expression (RegEx)

should be able to identify a symbol, we suggest to encode symbols with a special token in pre- or post-processing step. We could distinguish between integer numbers, floats, brackets etc.

Punct: The prediction of punctuation is also a simple task and thus the model achieved an F1-Score of 98%.

And: The model identified *and* labels with a precision of 77% and recall of 74%. Looking at the confusion matrix, it is not entirely clear for which labels the model has mistaken the *and* label, which implies that the mistakes were most likely distributed between a number of labels. This could also suggest, that the training data for this label is not entirely consistent.

OR: Although similar to the *and* label, the *or* label achieved a better performance, even though the amount of training data was lower.

Keywords for None Causal Segments (Key NC): The Key NC was predicted incorrectly in all cases, however, the training data only consisted out of 14 tokens and the test out of 4, so the results are not surprising. It might make sense, to merge the label with the *keywords for causal segments* label into one *keywords* label.

Condition, Variable, Statement, Cause, Effect, Cause Effect Relation, Separated Cause, Insertion, Negation: These labels were inferred with high precision and recall by the model, as can also be seen in the confusion matrix. There is no major difference in performance compared to the 300-dimensional random word embeddings model. The model clearly benefitted from the amount of training data for these labels, which was at minimum 172 tokens (Insertion label) and 12623 tokens for the condition label.

Separated Statement, Sentence: It seems like these two labels have benefitted the most from the BERT model. While the base (300-d) model was not able to predict the Separated Statement correctly at all, some improvements could be made with FastText and the BERT model achieved an F1-Score of 67%. Similar behavior was observed for the Sentence label. This is especially impressive due to the low amount of training data for those two labels, with only 27 Separated Statement tokens in the training dataset.

Separated Cause Effect Relation, Separated Negation, Separated None Causal, Separated Or, Separated Effect, Separated Effect, Separated Variable, Separated Condition: For these labels, there is no test data at all (Exception: 4 samples for the *Separated Cause Effect Relation* label; see Figure 4.2) and thus the performance on those labels could not be tested and cannot be evaluated. The training data is also so low, that it could make sense to simply omit these labels, or to merge them with other labels.

5.4 Improving Inference

In the preceding sections, we have evaluated our model for node label prediction (i.e. classification). Since our model does not explicitly train the tree parsing decoder, we will elaborate in this section how our model can also be used for inference and how the inference process can be improved. The naive approach is using a very simple greedy algorithm. Assuming we have a sentence with n tokens (a, b, c, d, e), the process is as followed:

1. Compute the softmax scores for all adjacent nodes $\{(a, b), (b, c), (c, d), (d, e)\}$

2. Concatenate the two nodes that achieved the highest posterior probability and update the list of adjacent nodes. In case c and d are concatenated to a parent representation p_1, the list of adjacent pairs is updated to $\{(a, b), (b, p_1), (p_1, e)\}$

3. Repeat until only one parent node is left. In this example, this would yield $\{p_4\}$.

This algorithm runs in linear runtime, however, there are some major drawbacks which can result in very poor results, especially for more complex and longer sentences. Figure 5.5 shows the output of the greedy algorithm and as we can see, specifically, the higher up structure and labels do not make much sense. For instance, there are two separated cause labels and a cause label for the segment "if set to" although it should be "if set to true". The reason for major mistakes like that can be explained through error propagation, since at each parsing step, the decoder relies on the preceding steps to be correct. We consider two ways of optimizing the models' inference for such sentences.

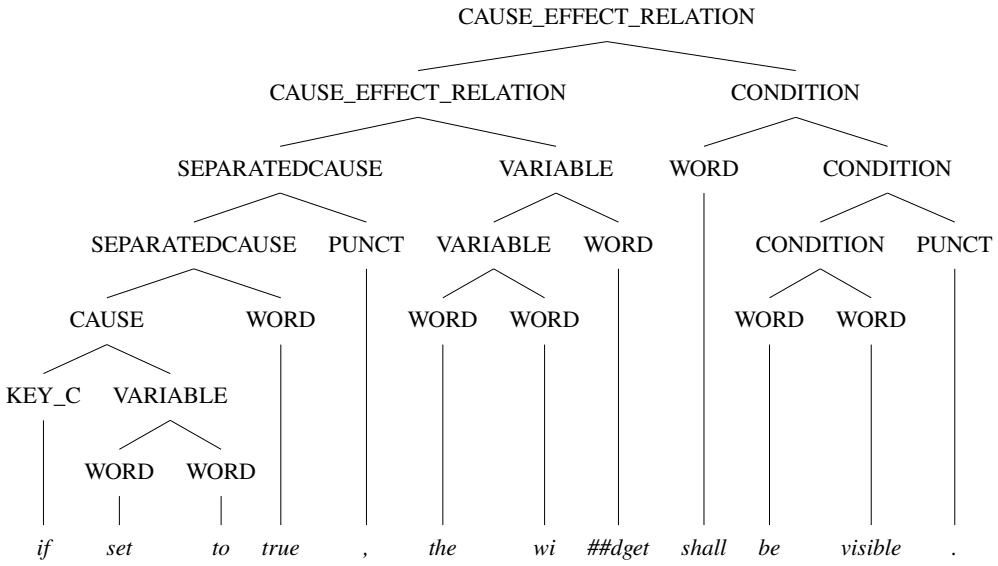

Figure 5.5.: Naive approach

5.4.1 Temperature Scaling

The first problem lies within the functionality of the softmax function. Contrary to popular belief, the softmax function's output should not be interpreted as a confidence score as the predicted probabilities tend to be too high, even if the input does not make any sense [18]. One solution would be to introduce an additional class, that labels nodes that should not be concatenated as $UNKNOWN$ or similar. However, therefore we would need to expand the datasets with samples of bad parsing decisions, which is outside the scope of this thesis. Guo et al. [18] suggests doing a post-training calibration, called temperature scaling (TS), which complements the neural network

with a parameter T in the softmax function, with the objective of achieving confidence scores that more closely represent the "certainty/uncertainty" of the model.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{28} \exp(\frac{x_j}{T})} \tag{5.1}$$

We have implemented TS into our model and compare the results and impact of TS after the next subsection.

5.4.2 Beam Search

The second problem is the inability of the tree parsing algorithm to correct mistakes (i.e. error propagation). In many cases, the model only detects at a later stage that the underlying sub-tree is incorrect, since the softmax probabilities are decreasing. The total number of all possible trees can be described by the Catalan number $C_k = \frac{1}{k} \cdot \binom{2\cdot(k-1)}{(k-1)} = \frac{(2\cdot(k-1))!}{k!\cdot(k-1)!}$, where k is the number of tokens or non-terminal nodes. Leveraging dynamic programming, by using the CYK algorithm to find the optimal tree, would still leave us with cubic runtime. For a better runtime in most cases, we can use the *beam search* algorithm, which is more greedy than the CYK algorithm but less greedy than the naive approach. Beam search functions by caching a predefined number (beam width n) of most promising nodes in a set. In detail, the algorithm works like this (See Alg. 1 for pseudocode):

1. Compute the softmax of all neighboring nodes $\{(a,b),(b,c),(c,d),(d,e)\}$

2. Sort the results by total confidence, i.e. the confidence of the node itself and all children, and cache the n best results.

3. Repeat step 1 and 2 with the new cached nodes, for example with (a,b,p'_1,e) and (p''_1,c,d,e). We once again only save the n best results.

4. Once there are only single nodes left in the cache set, we chose the tree with the highest total confidence score.

The results of the parser have improved significantly using beam search, but we have to keep in mind that it comes at the cost of runtime, and it still does not guarantee the best result.

Algorithm 1 Beam Search Parser

```
tokens = tokenize(sentence)
nodes = []
for token in tokens do
    nodes.append(new Node from token)
end for
predictions = [{
    nodes: nodes,
    total_confidence: 0,
    parent_nodes: []
} ]
while step < length(nodes) do
    new_predictions = []
    for prediction in predictions do
        parent_nodes = []
        compute output vector for each node in prediction[nodes]
        add adjacent nodes as parent node to parent_nodes and compute the softmax
        for parent_node in parent_nodes do
            Create new_prediction from parent_node
            if prediction does not already exist then
                predictions.append(new_prediction)
            end if
        end for
    end for
    sort(new_predictions)
    predictions = new_predictions[0:beam_width]        ▷ Only save the beam_width best predictions
    step = step + 1
end while
return prediction with highest total confidence
```

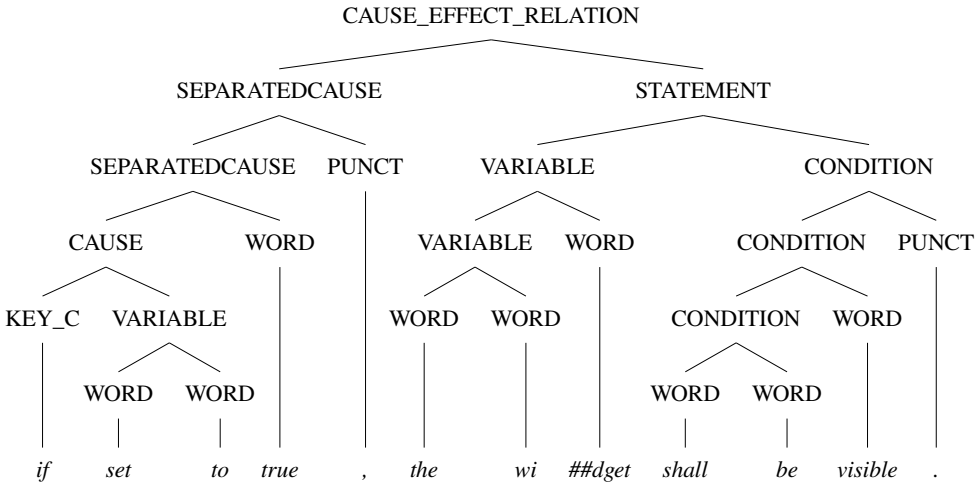

Figure 5.6.: Beam search with beam width of 5

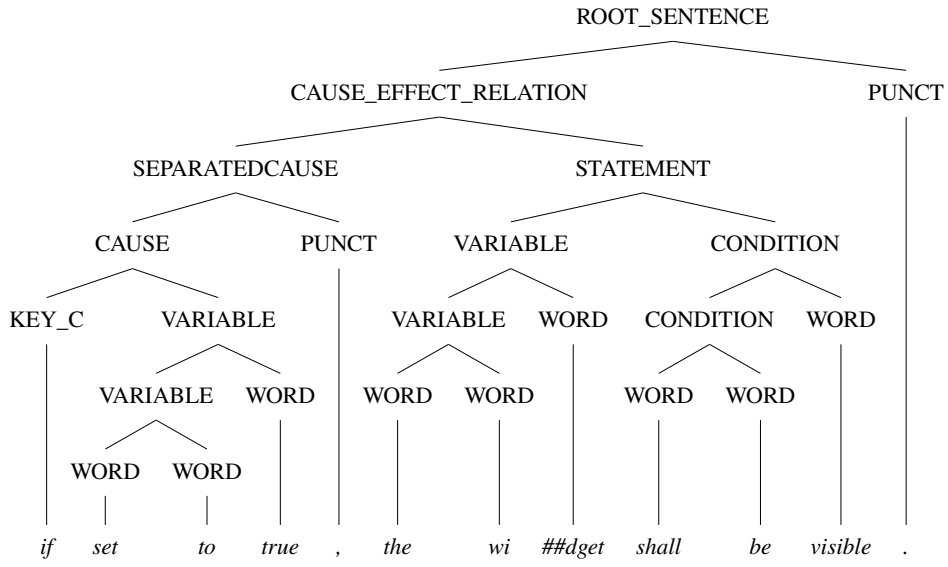

Figure 5.7.: Beam search with beam width of 10

The above figures show increasing beam width can further increase the output of our model, although even a beam width of 5 results in a better outcome than the naive approach. The runtime increases lineally with the beam width and sentence size, but setting the beam width too high for longer sentences might result in *Out-Of-Memory* or similar errors. We can also see the effect of temperature scaling by comparing Figure 5.7 to Figure 5.8 which was computed without temperature scaling. Clearly, the inferred tree with TS is better than the one without.

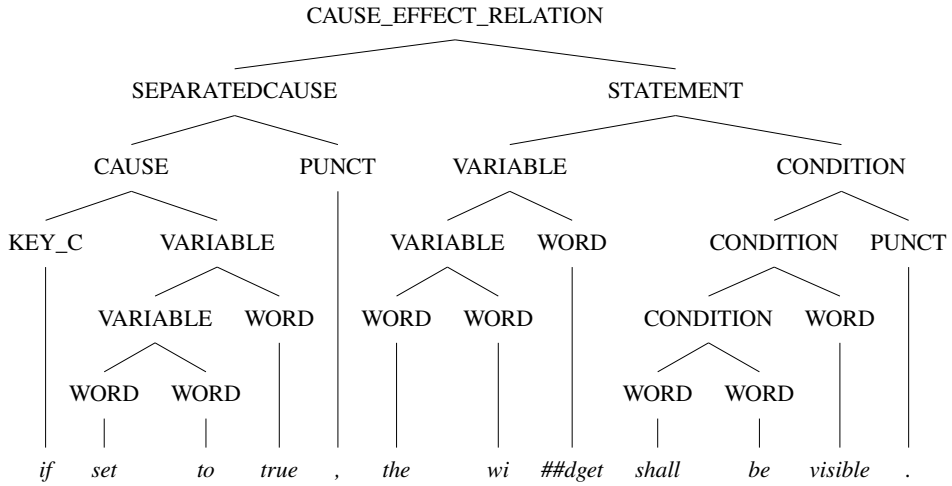

Figure 5.8.: Beam search with beam width of 10 without temperature scaling

5.4.3 Other improvements

Further improvements that can be made are rule-based adjustments. As mentioned before, the BERT tokenizer sometimes splits one word into two tokens, like the word "widget" in the examples above. We know that normally the tokens "wi" and "##dget" belong together, but they are not merged correctly. In this case, we can either introduce a rule during inference to always merge those kinds of tokens first, or we can merge the tokens in a post-processing step after the actual inference has been done. Similar rules can be introduced, for instance, a rule that a *Root Sentence* must only occur at the root node and has a *Punt* node as the right child. These rules will then prevent mistakes from happening in the first place during inference, and the beam width and computing time can be reduced.

Demo: CATE

6

To illustrate the results of our experiments, we have built the online demo *CATE* (acronym for **CA**usality **T**ree **E**xtractor), accessible at `causalitytreeextractor.com`.

User Interface The UI consists of two parts. The left side features a text input field, where a causal sentence can be entered. In addition, the configuration of *CATE* can be adjusted. It is possible to set the beam width and to select whether temperature scaling should be used for prediction or not. Additionally, the user can specify whether the binary tree should be built using left or right branching and which word embeddings should be utilized. After clicking on the predict button, the parsing result is displayed on the right side. The user can interact with the binary tree by expanding or collapsing certain segments. In addition, the time of the prediction request is displayed at the top.

Technologies The backend is built with *Flask*, a web framework for Python and hosted via nginx and Gunicorn on the same Linux machine that was used for training. Styling was done with the *Tailwind CSS* framework, and the illustration of the trees was implemented using the *GoJS* Framework.

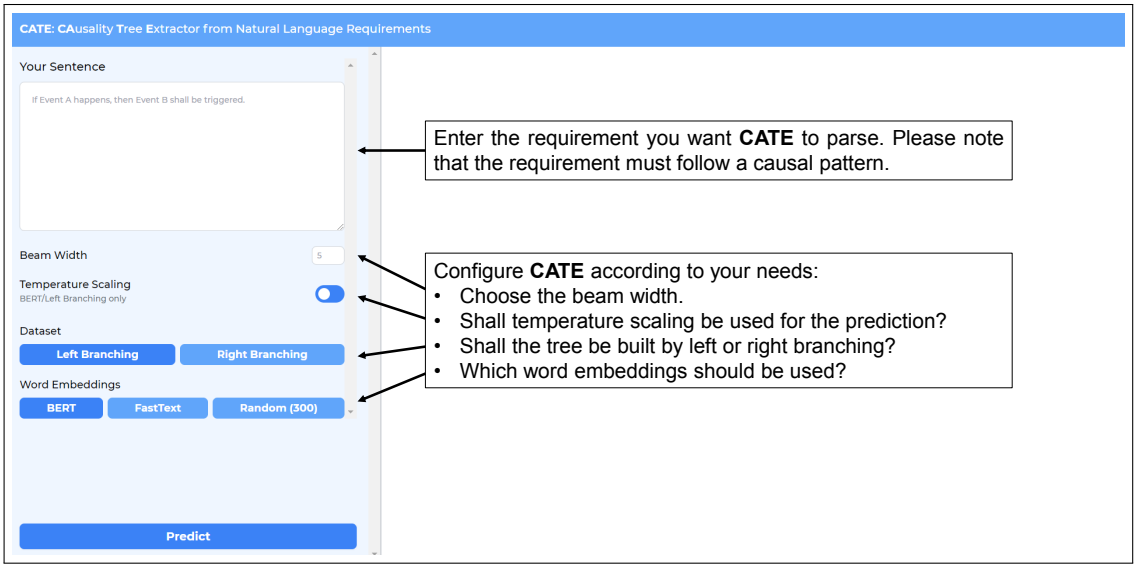

Figure 6.1.: Configuration options of *CATE*.

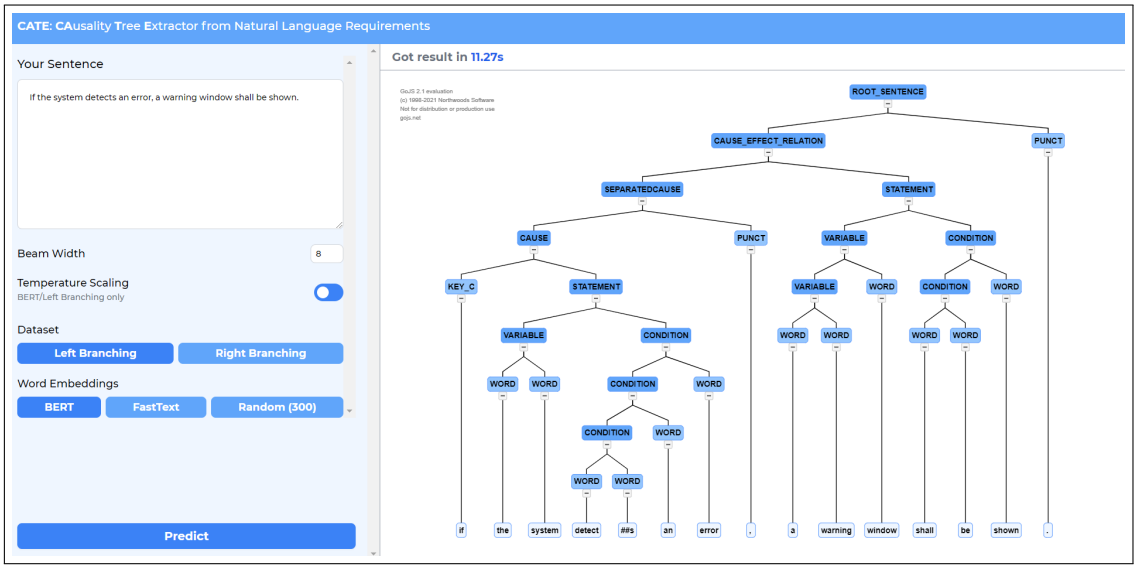

Figure 6.2.: Binary parse of the requirement "*If the system detects an error, a warning window shall be shown.*"

Conclusion

7

7.1 Summary

In this paper, we have shown how to leverage and fine-tune recursive neural networks to extract causality from natural language requirements. Several experiments have been conducted in order to improve accuracy, recall and precision on our dataset, and we have additionally explored ways to improve inference on new sentences. Our experiments and research have shown that utilizing a left-branching dataset, in combination with BERT word embeddings, have achieved the best results with an F1-Score of 94.07%. For inference, we have observed significant improvements by using a beam-search algorithm and calibration of confident scores with the help of temperature scaling. Additionally, a web demo was built to showcase our research with the option of switching between models and inference configurations that were examined.

7.2 Future Work

There are a few limitations in the model, that could be addressed in future work.

(1) Currently, the biggest limitation is the computation time and performance during inference. Although the model performs reasonably well with shorter sentences and a relatively small beam width (< 10), it fails to do so once sentences become longer and more complex. To improve the results, it is possible to increase the beam width, however, by doing so we have experienced memory related errors. As a consequence, high-performance computers or servers are needed to run the model in an adequate time, potentially resulting in high costs. One practical solution would be to select the beam width dynamically, based on the sentence length and computer performance, guaranteeing a result in an acceptable time, but this could still lead to misparsing of the sentence. As a potential solution, we propose another approach that uses a pre-processing step to split larger sentences into two or more segments, which are later merged back together. Let us consider the sentence “If the content is not available in the detected system language, English is selected by default.”. By fine-tuning BERT for token classification [21], we can split the sentence into the segments, cause “If the content is not available in the detected system language” and statement/effect “English is selected by default”. For each of those segments, we can apply the RNN to get better results in a shorter time. Now we just need to merge both sub-trees with the punctuation, and we get the final tree.

(2) The current model was trained on a simple recursive neural network, yet improvements were made with an extension of the normal RNN. In [42], Socher et al. propose a Recursive

Neural Tensor Network (RNTN) that has performed significantly better than the standard RNN. By introducing an additional neural tensor layer in the composition function, the child nodes can interact more explicitly with the parent and not just implicitly through the non-linearity function that "squashes" two vectors into one. Other RNN implementations suggest to also pass the k left and right adjacent nodes of each child node to the composition function, in order to make the model context-aware.

(3) We have only compared contextual to traditional and random word embeddings which suggested that contextual word embeddings can contribute to significant improvements, however, other contextual word embeddings than BERT could also be used. In future work, we can explore performance of BERT word embeddings compared to ELMo [37] and Flair [2]. The Flair framework [3] also allows for stacking of different word embeddings, like GloVe and ELMo word embeddings, potentially further improving results.

(4) Last, but not least, to derive test cases it might be of importance to know whether cause and effect are connected by a logical implication ($cause \implies effect$) or a logical equivalence ($cause \iff effect$). These kinds of dependencies can be illustrated by a dependency tree, which can be transformed from constituency trees through another model. In future work, we can explore how we develop such a model or even how to train a model jointly for constituency and dependency parsing.

General Addenda

A

A.1 Dataset Labels

1. **Root Sentence:** represents the root node of a sentence
2. **Symbol:** used to mark special symbols
3. **Punct:** used to indicate punctuation marks
4. **And:** used to annotate conjunctions
5. **Or:** used to annotate disjunctions
6. **Key-C:** highlights certain cue phrases that indicate causality
7. **Key-NC:** highlights certain cue phrases that indicate non-causal segments in a sentence
8. **Condition:** indicates the verb phrases that belong to a variable
9. **Variable:** indicates noun phrases in a sentence
10. **Statement:** combination of condition and variable segments
11. **Cause:** indicates a cause segment
12. **Effect:** indicates an effect segment
13. **Cause Effect Relation:** captures all segments that belong to the causal relation
14. **Separated Cause**
15. **Insertion:** used to annotate any kinds of insertions in a sentence (e.g., bracket expressions that are not essential for the interpretation of a sentence but provide additional information)
16. **Negation:** indicates negations (e.g. negated conditions)
17. **None Causal:** indicates segments that are not part of the causal relation
18. **Separated Statement**
19. **Separated And:** that are syntactically separated from other and fragments
20. **Sentence:** used to connect, e.g., non-causal and causal segments of a sentence
21. **Separated Cause Effect Relation**
22. **Separated Negation**
23. **Word:** is distributed at the bottom level of the tree and assigned to the individual words of a sentence
24. **Separated None Causal**

25. **Separated Or**
26. **Separated Effect**
27. **Separated Variable**
28. **Separated Condition**

Separated... labels are used to highlight self-contained text fragments that are syntactically separated from other fragments. The label is needed, for example, when a Statement segment will be merged with a comma token. The comma turns the segment into a Separated Statement. [13]

Figures

B

B.1 Confusion matrices

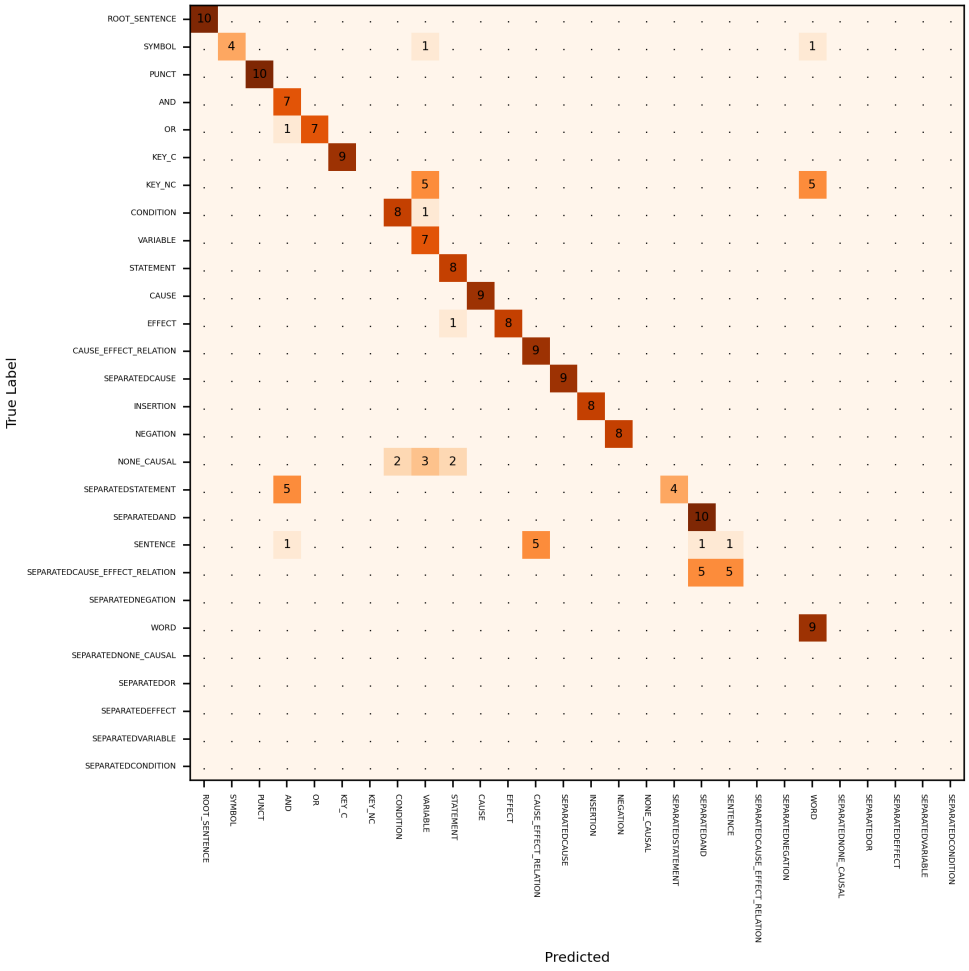

Figure B.1.: Left-branching random (30 dimensions) model Confusion Matrix

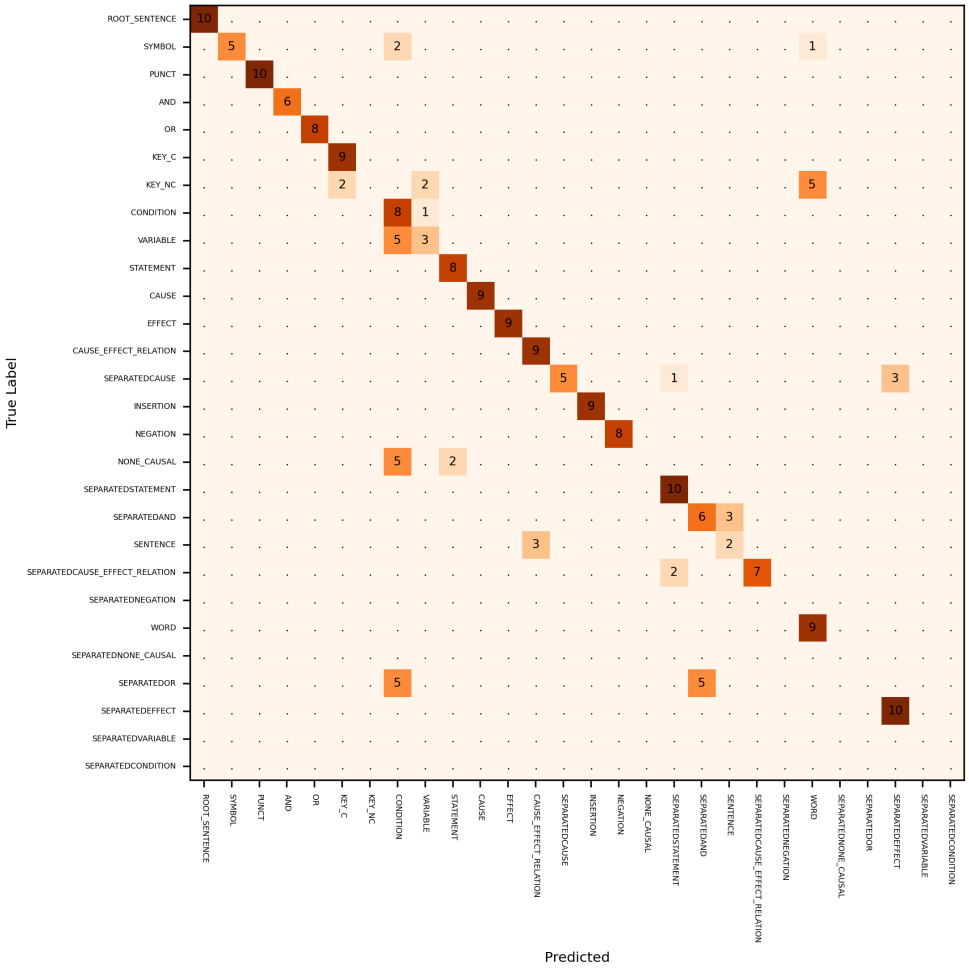

Figure B.2.: Right-branching random (30 dimensions) model Confusion Matrix

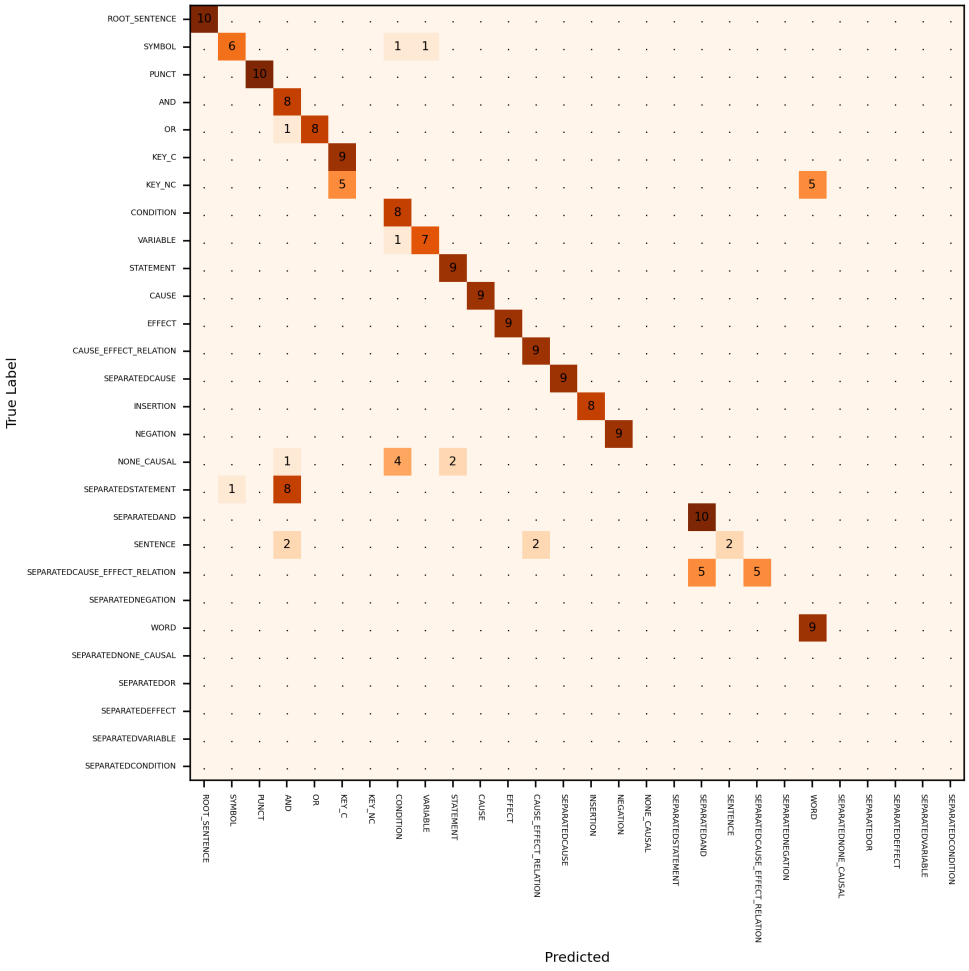

Figure B.3.: Left-branching random (60 dimensions) model Confusion Matrix

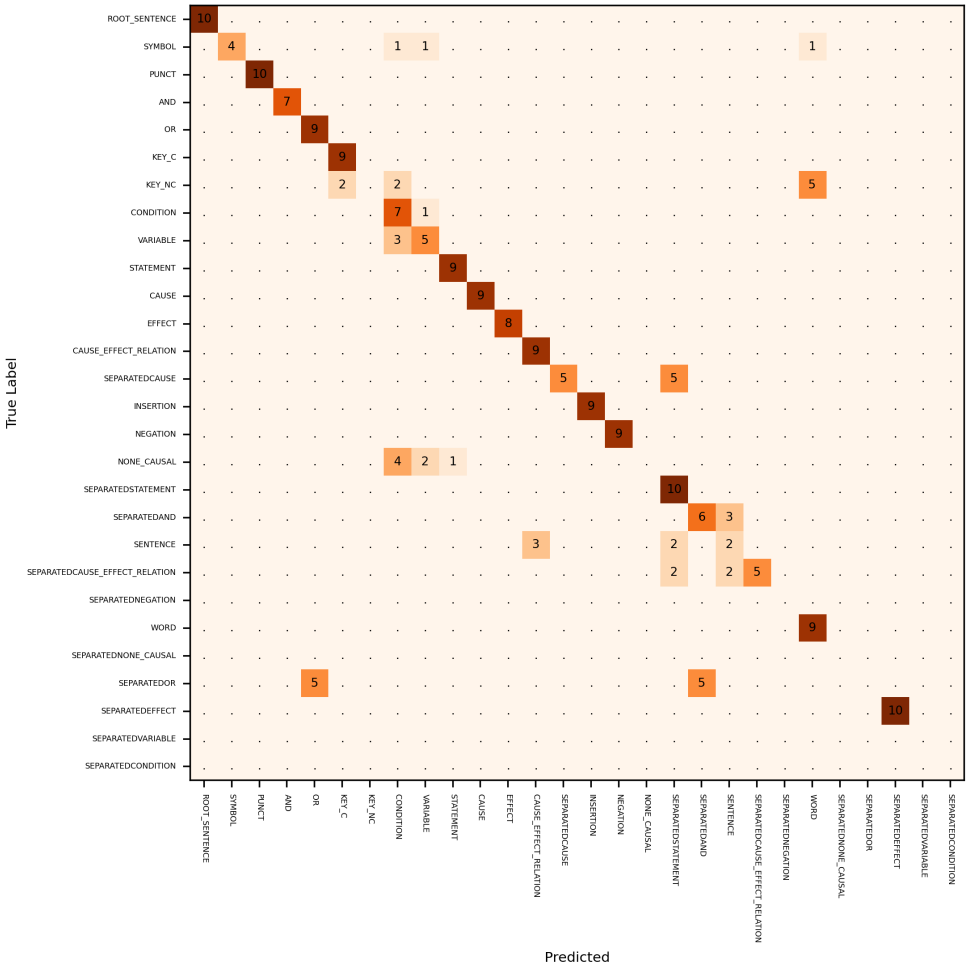

Figure B.4.: Right-branching random (60 dimensions) model Confusion Matrix

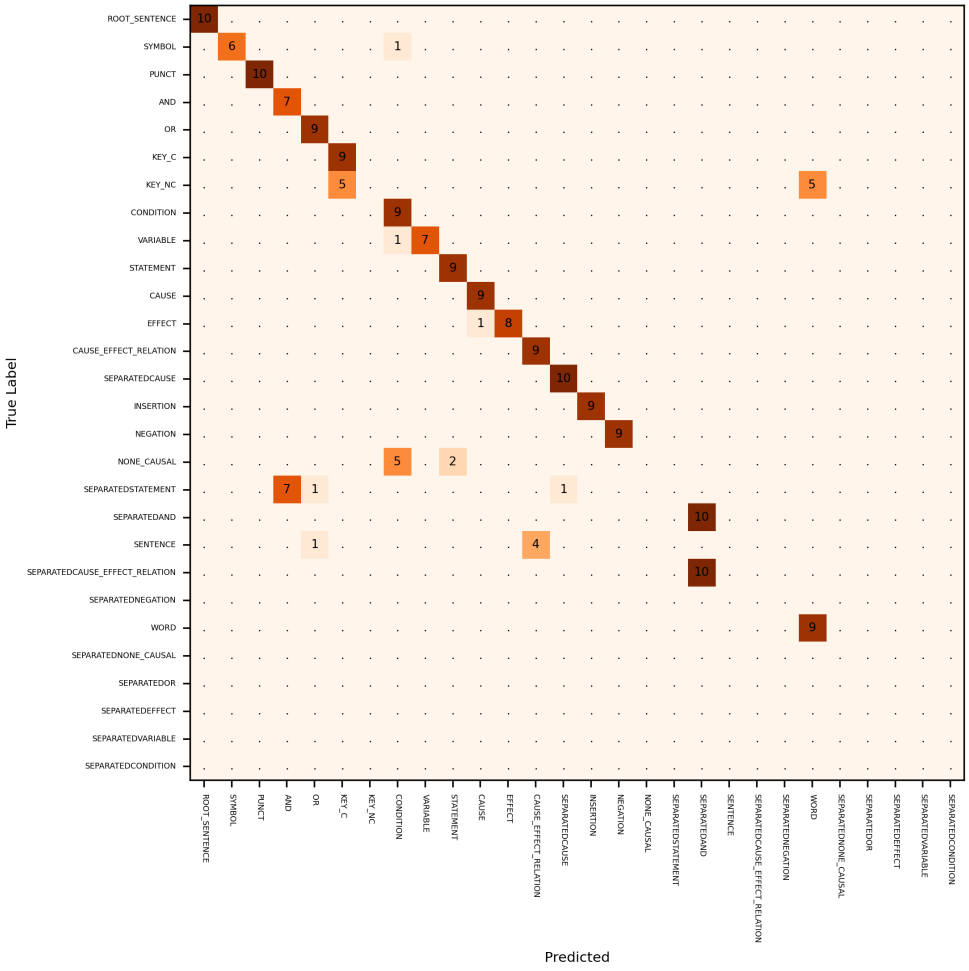

Figure B.5.: Left-branching random (300 dimensions) model Confusion Matrix

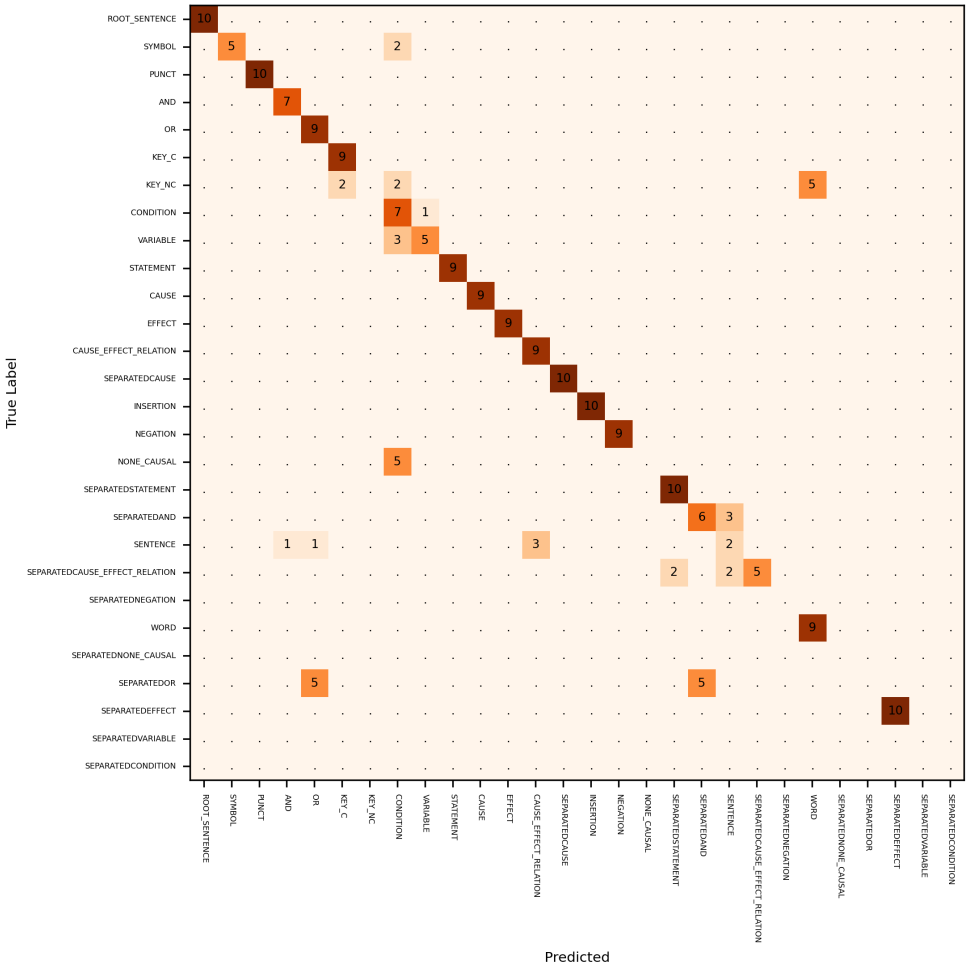

Figure B.6.: Right-branching random (300 dimensions) model Confusion Matrix

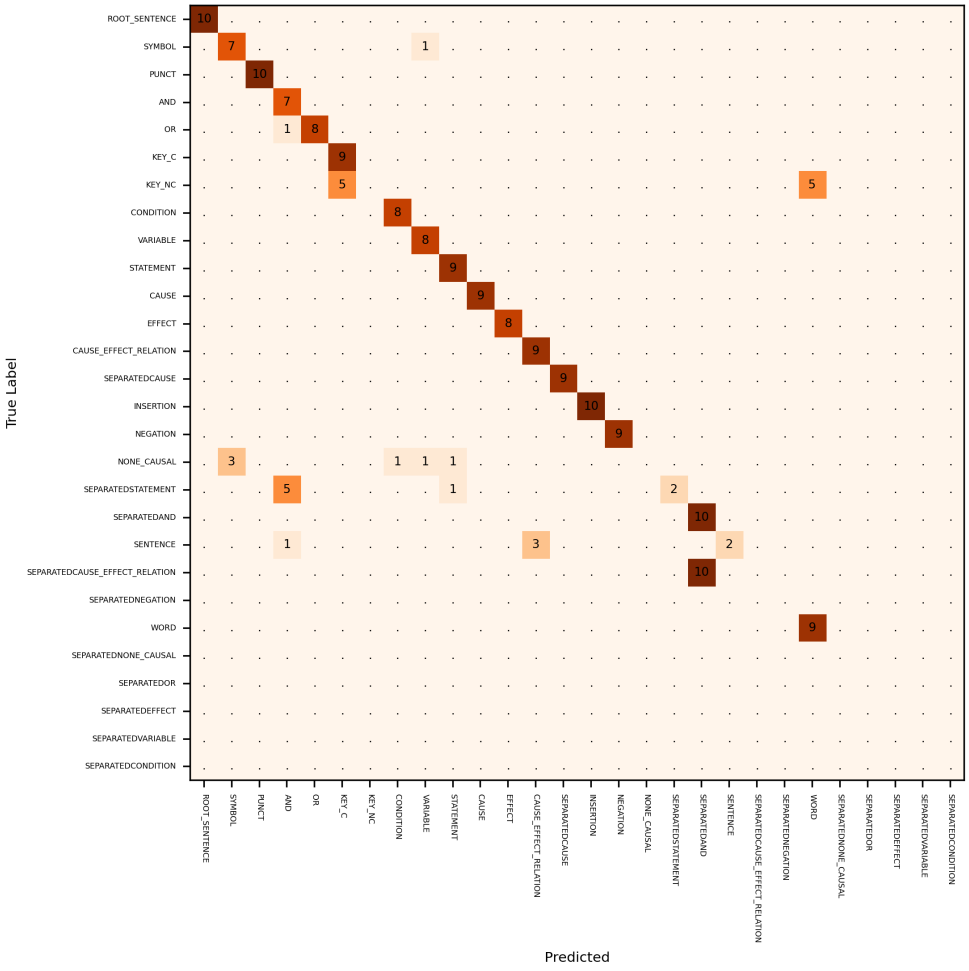

Figure B.7.: Left-branching GloVe model Confusion Matrix

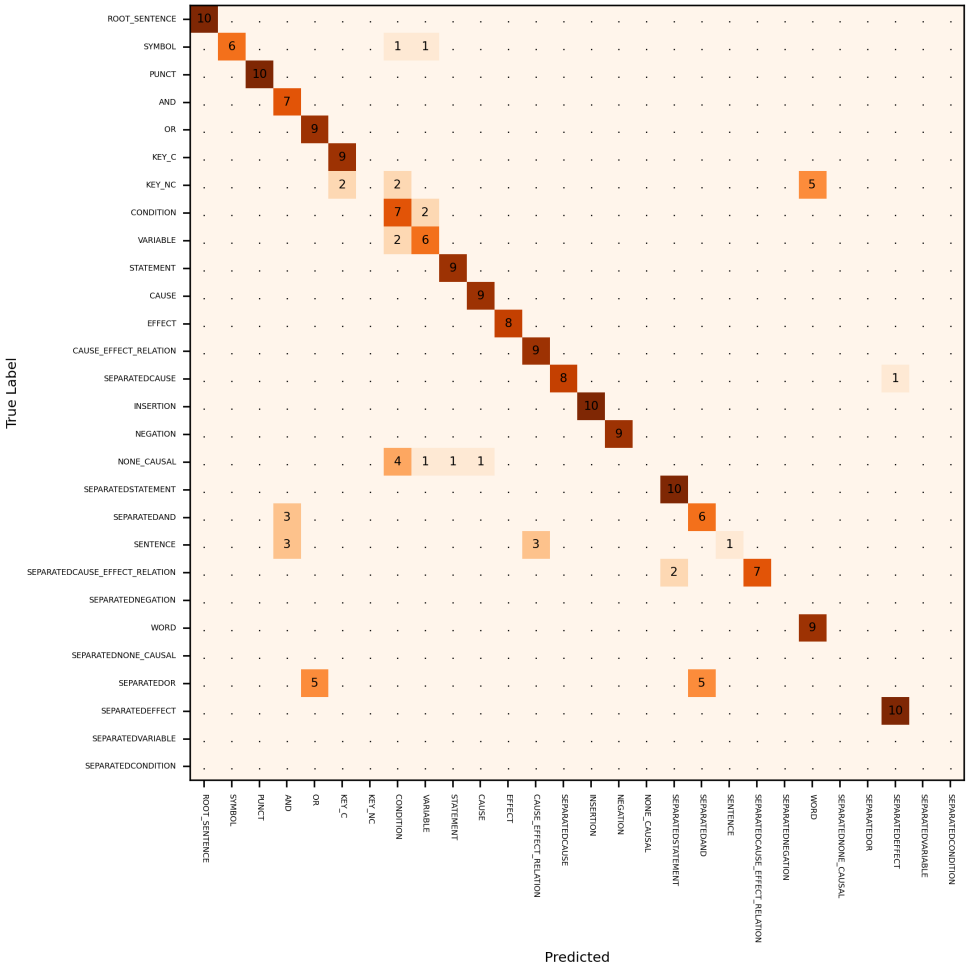

Figure B.8.: Right-branching GloVe model Confusion Matrix

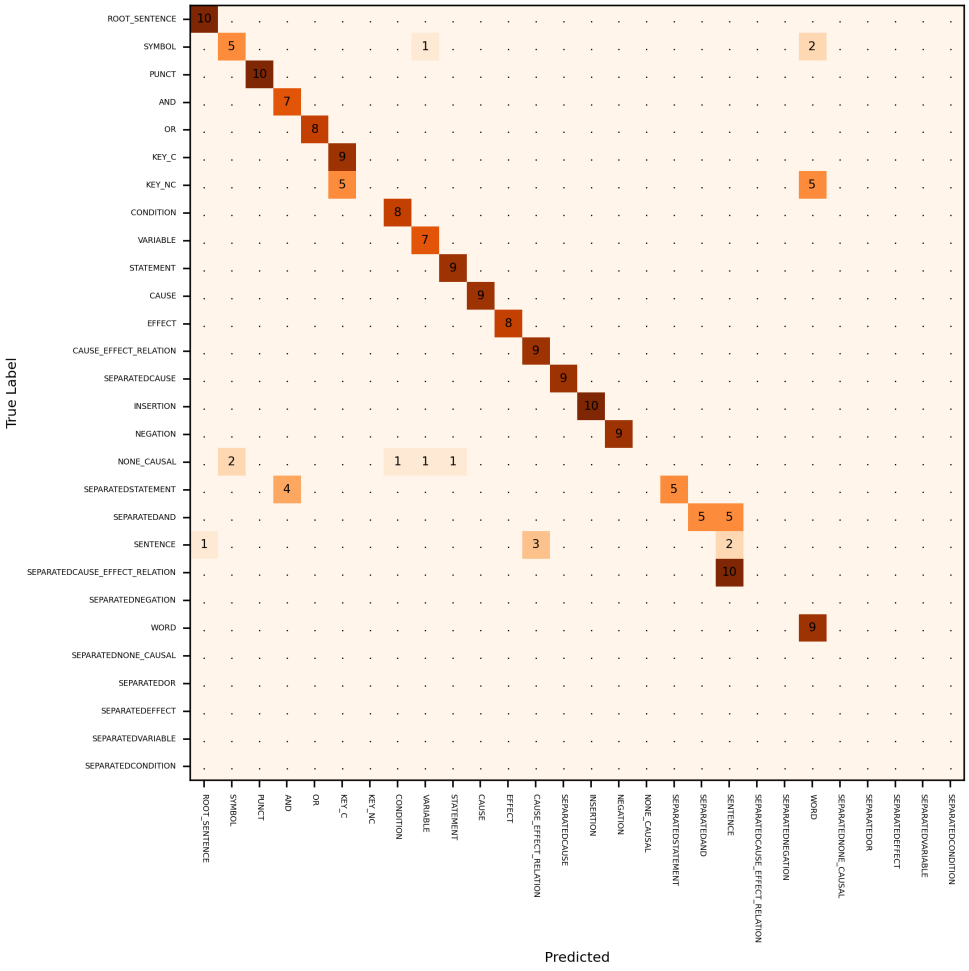

Figure B.9.: Left-branching FastText model Confusion Matrix

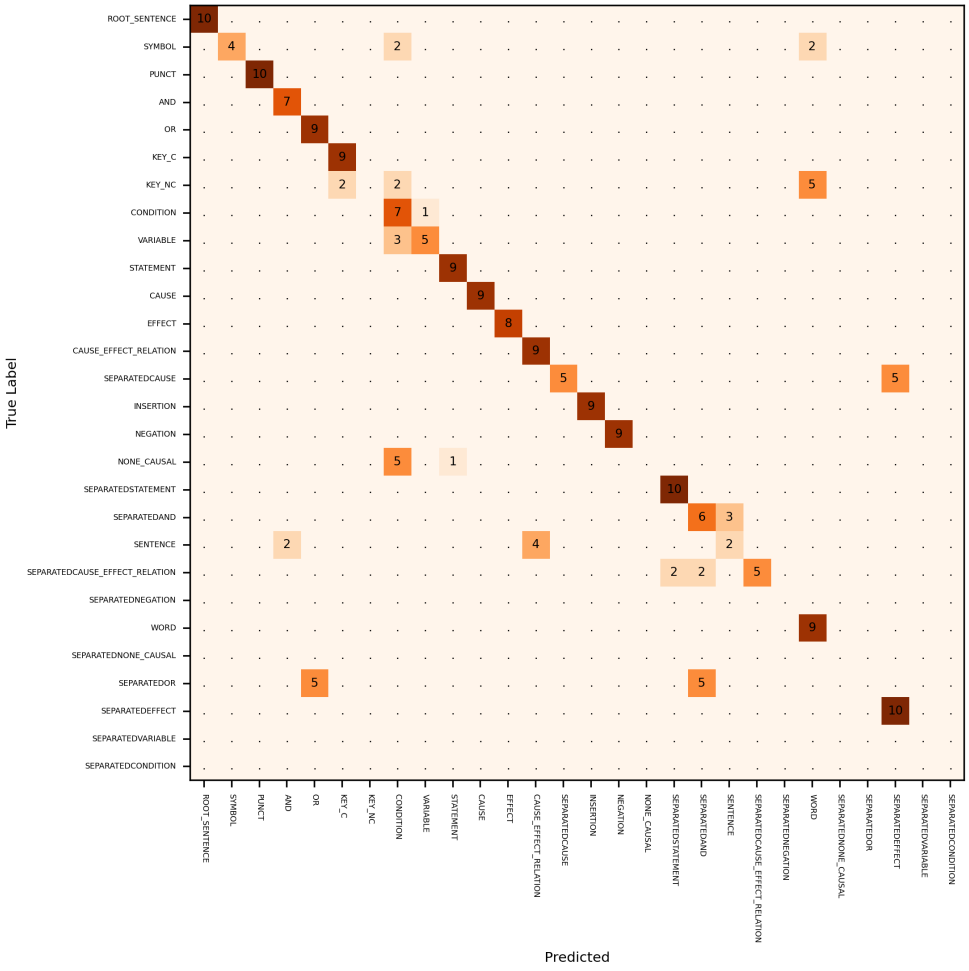

Figure B.10.: Right-branching FastText model Confusion Matrix

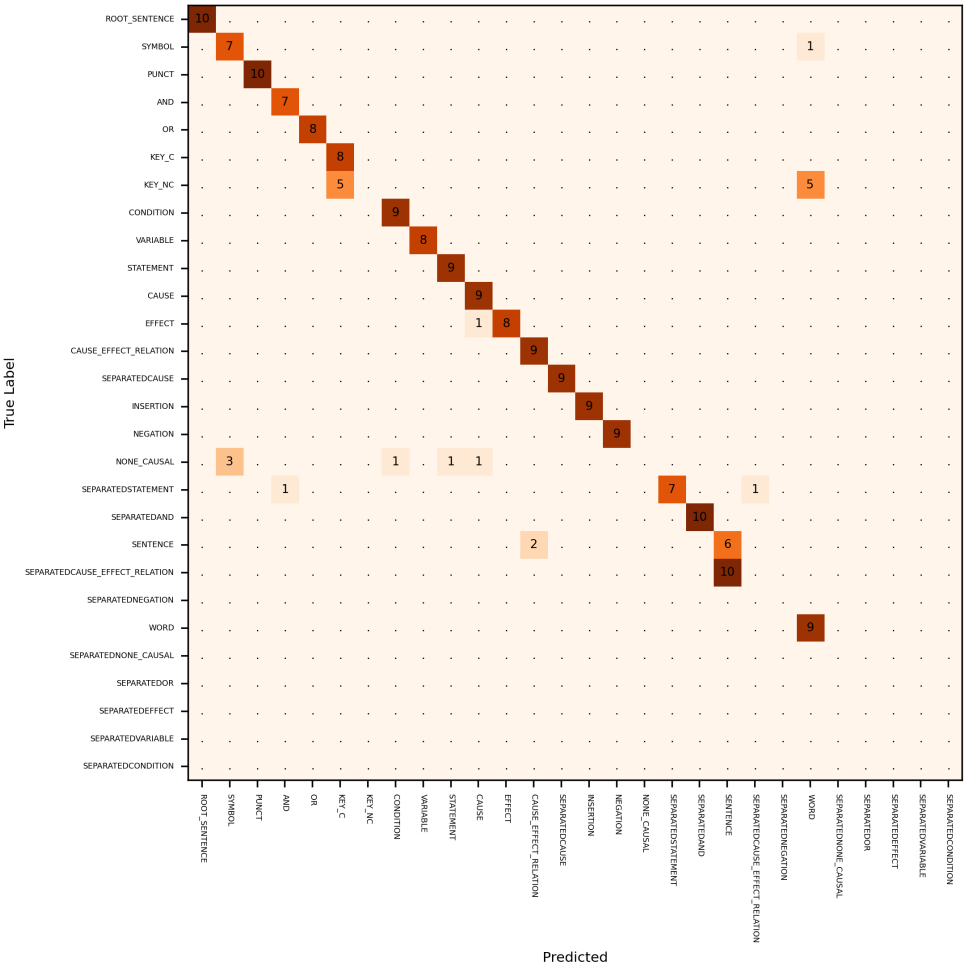

Figure B.11.: Left-branching BERT model Confusion Matrix

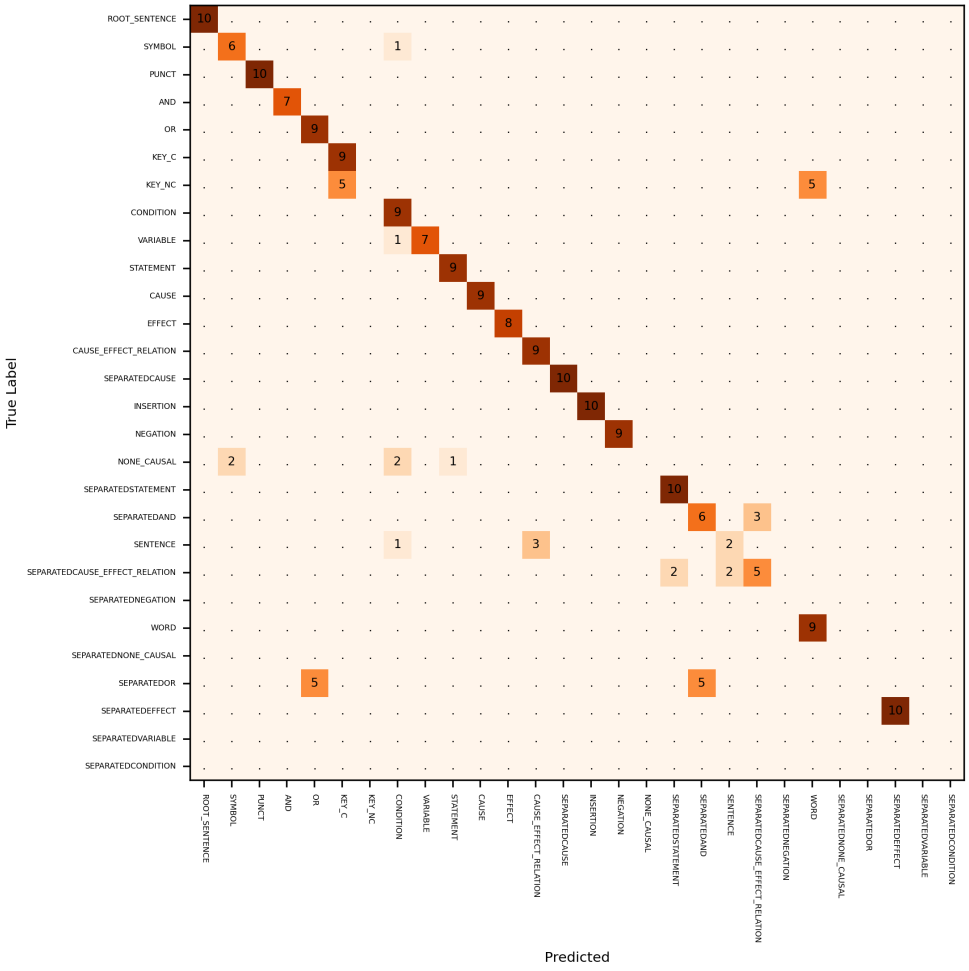

Figure B.12.: Right-branching BERT model Confusion Matrix

B.2 Classification reports

Figure B.13.: Left-branching random (30 dimensions) model Classification Report

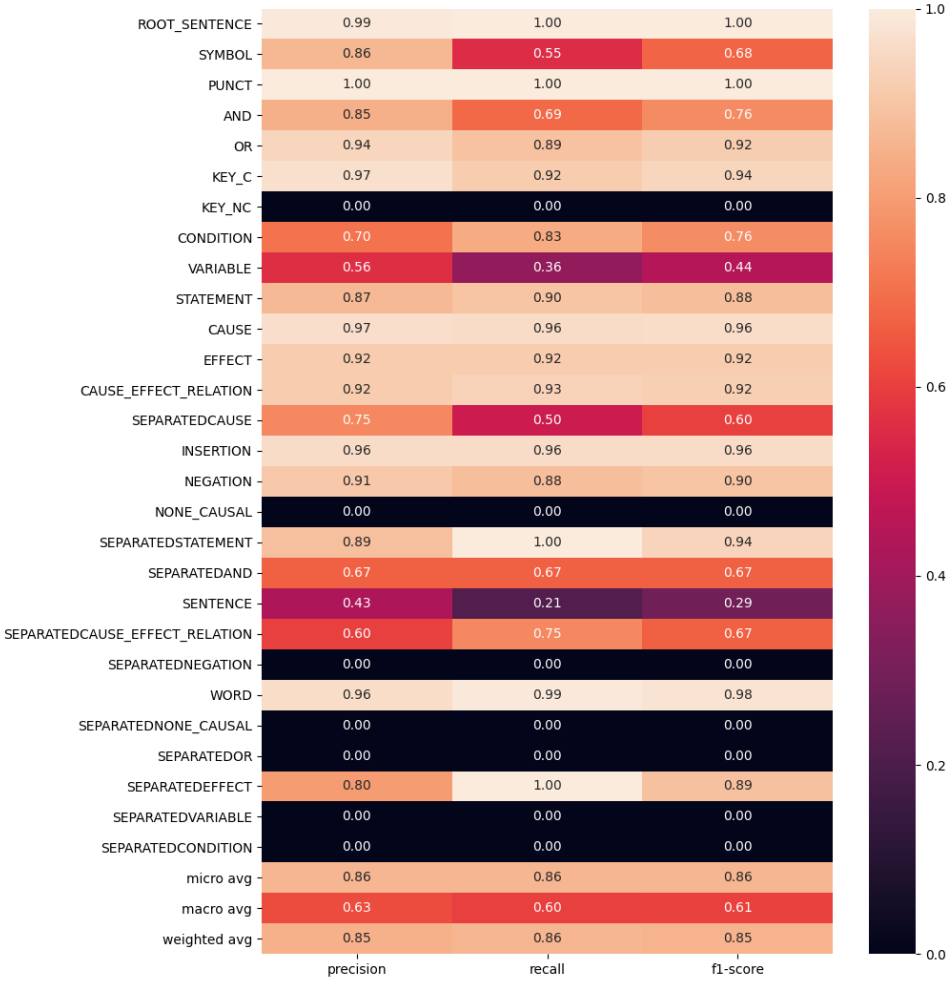

Figure B.14.: Right-branching random (30 dimensions model Classification Report

Figure B.15.: Left-branching random (60 dimensions) model Classification Report

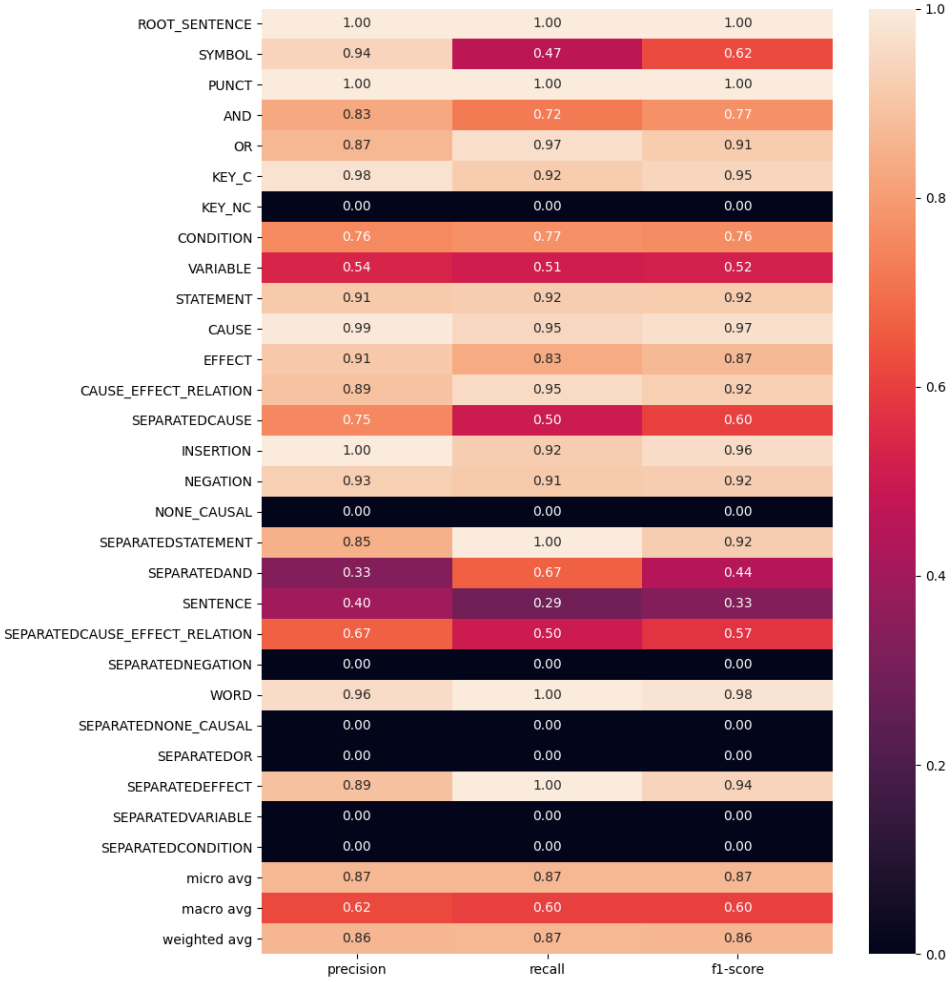

Figure B.16.: Right-branching random (60 dimensions model Classification Report

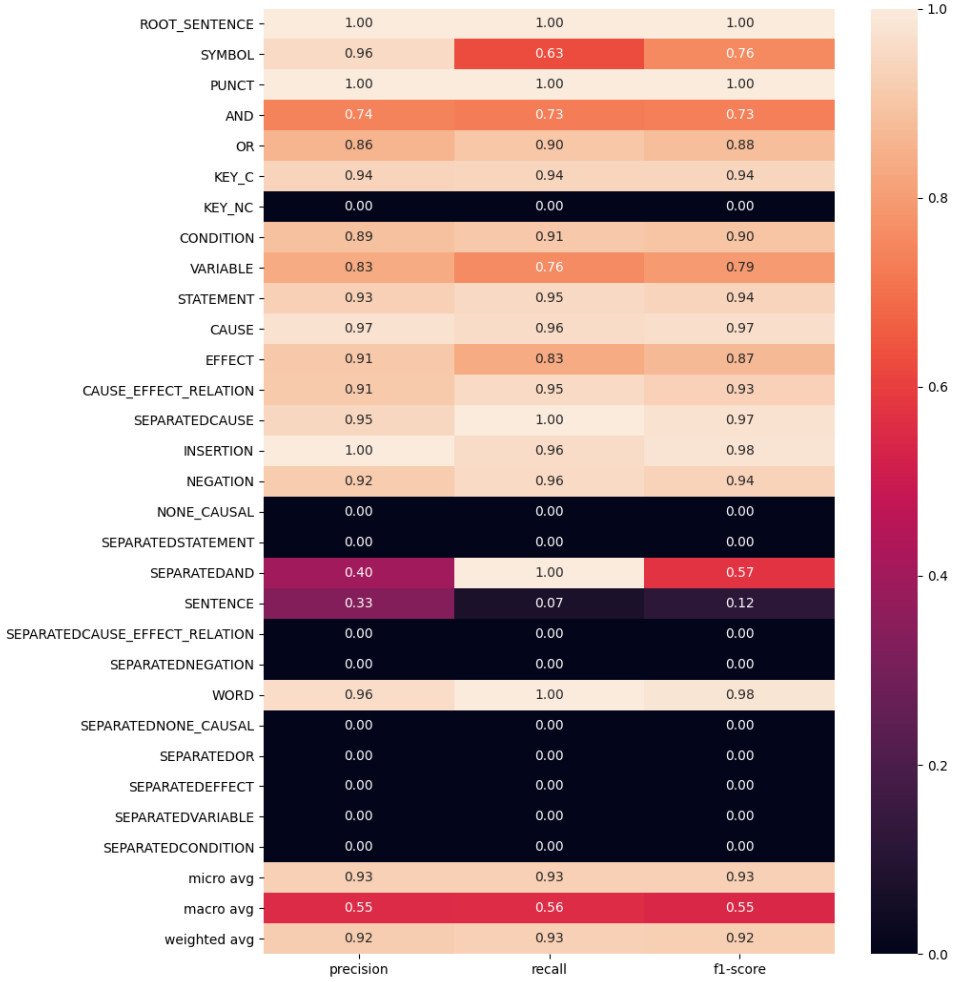

Figure B.17.: Left-branching random (300 dimensions) model Classification Report

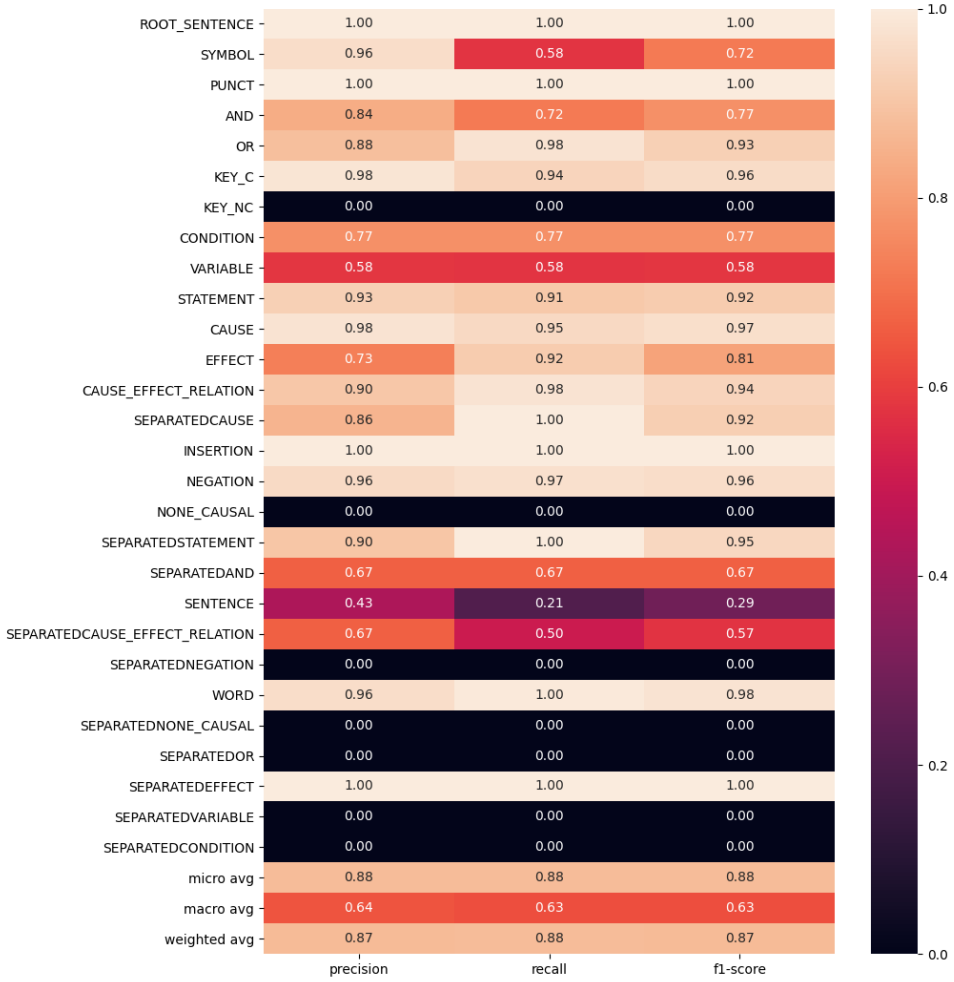

Figure B.18.: Right-branching random (300 dimensions model Classification Report

Figure B.19.: Left-branching GloVe model Classification Report

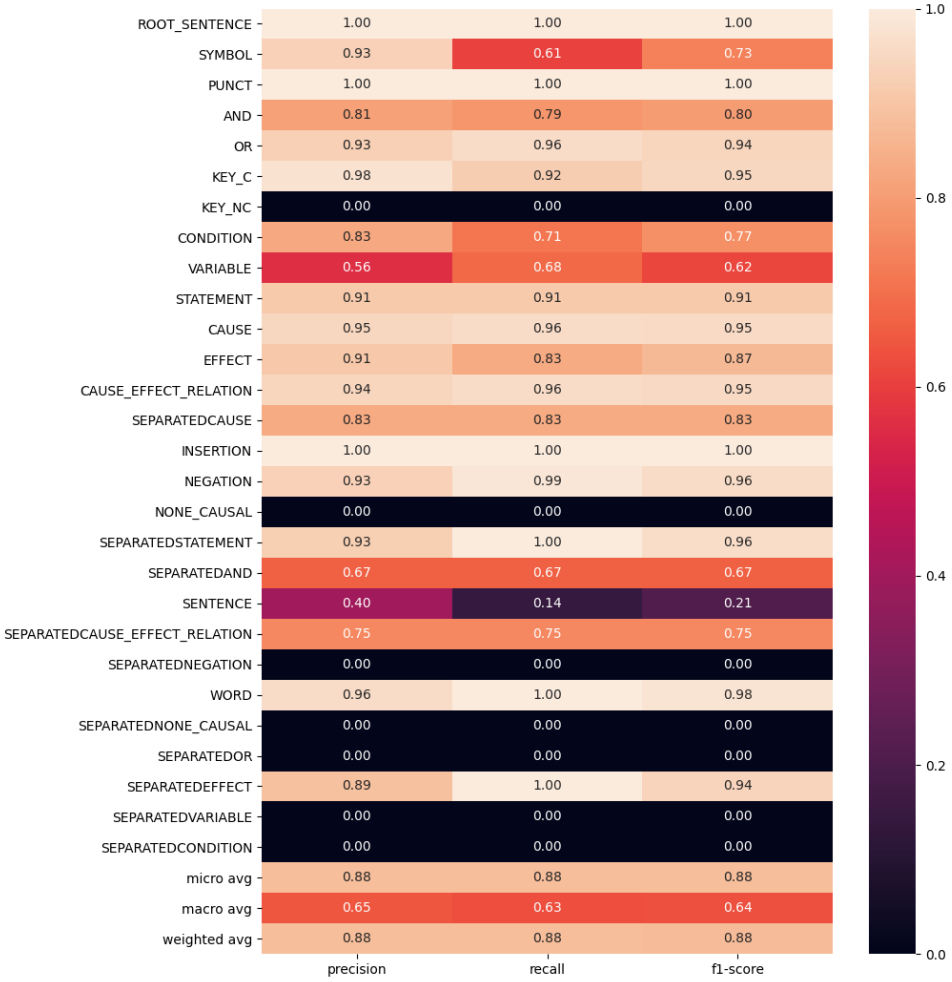

Figure B.20.: Right-branching GloVe model Classification Report

Figure B.21.: Left-branching FastText model Classification Report

Figure B.22.: Right-branching FastText model Classification Report

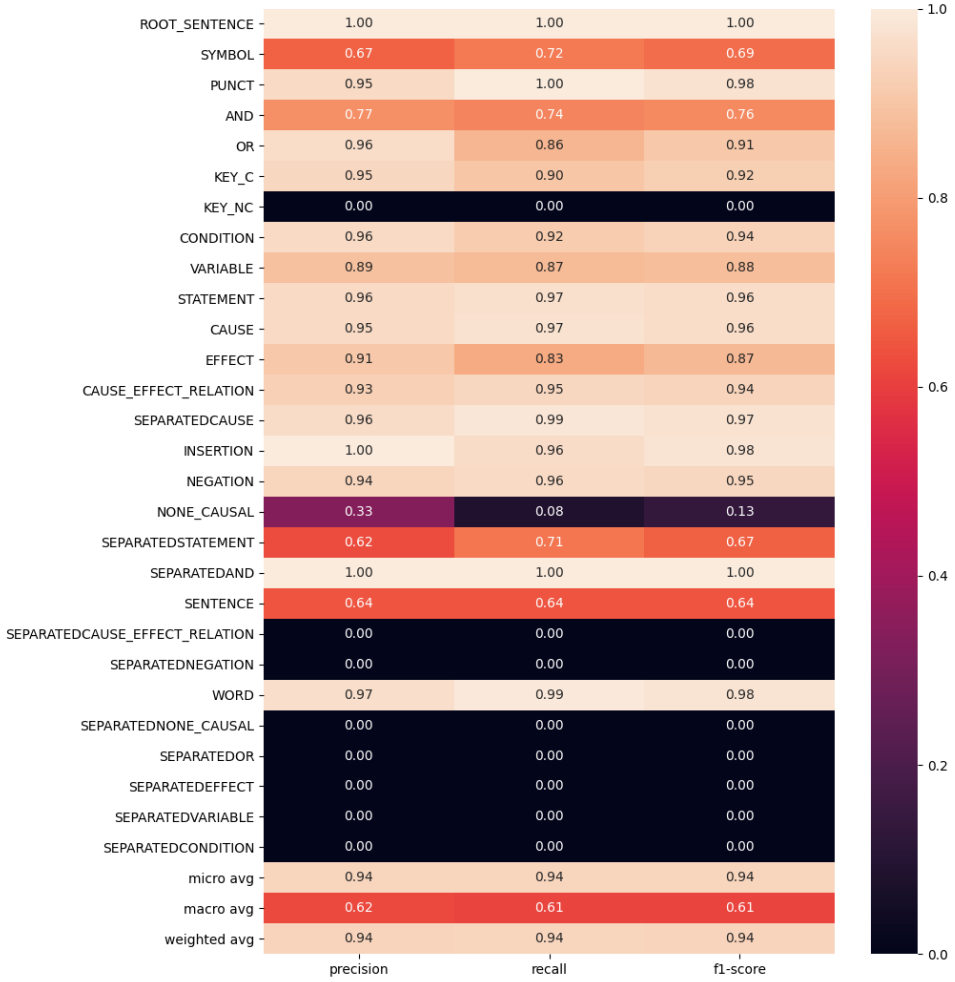

Figure B.23.: Left-branching BERT model Classification Report

Figure B.24.: Right-branching BERT model Classification Report

List of Figures

List of Tables

Bibliography

[1] *A brief history of Natural Language Processing*. URL: https://www.cs.bham.ac.uk/~pjh/sem1a5/pt1/pt1_history.html (visited on 07/22/2021).

[2] A. Akbik, D. Blythe, and R. Vollgraf. "Contextual String Embeddings for Sequence Labeling". In: *COLING 2018, 27th International Conference on Computational Linguistics*. 2018, pp. 1638–1649.

[3] A. Akbik et al. "FLAIR: An easy-to-use framework for state-of-the-art NLP". In: *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*. 2019, pp. 54–59.

[4] E. Blanco, N. Castell, and D. I. Moldovan. "Causal Relation Extraction." In: *Lrec*. Vol. 66. 2008, p. 74.

[5] J. Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU)*. 2020. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ (visited on 07/18/2021).

[6] E. Charniak. "A maximum-entropy-inspired parser". In: *1st Meeting of the North American Chapter of the Association for Computational Linguistics*. 2000.

[7] M. Collins. "Three Generative, Lexicalised Models for Statistical Parsing". In: *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*. ACL '98/EACL '98. Madrid, Spain: Association for Computational Linguistics, 1997, pp. 16–23. DOI: 10.3115/976909.979620. URL: https://doi.org/10.3115/976909.979620.

[8] J. Cross and L. Huang. "Incremental Parsing with Minimal Features Using Bi-Directional LSTM". In: *CoRR* abs/1606.06406 (2016). arXiv: 1606.06406. URL: http://arxiv.org/abs/1606.06406.

[9] M.-C. De Marneffe, B. MacCartney, C. D. Manning, et al. "Generating typed dependency parses from phrase structure parses." In: *Lrec*. Vol. 6. 2006, pp. 449–454.

[10] J. Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[11] C. Dyer et al. "Recurrent Neural Network Grammars". In: *CoRR* abs/1602.07776 (2016). arXiv: 1602.07776. URL: http://arxiv.org/abs/1602.07776.

[12] e. a. Falcon WA. "PyTorch Lightning". In: (2019). URL: https://github.com/PyTorchLightning/pytorch-lightning.

[13] J. Fischbach et al. “Fine-grained causality extraction from natural language requirements using recursive neural tensor networks”. In: *arXiv preprint arXiv:2107.09980* (2021).

[14] J. Fischbach et al. “Towards Causality Extraction from Requirements”. In: *2020 IEEE 28th International Requirements Engineering Conference (RE)* (2020). DOI: 10.1109/re48521.2020.00053.

[15] R. Girju et al. “Semeval-2007 task 04: Classification of semantic relations between nominals”. In: *Proceedings of the Fourth International Workshop on Semantic Evaluations (SemEval-2007)*. 2007, pp. 13–18.

[16] C. Goller and A. Küchler. “Learning Task-Dependent Distributed Representations by Backpropagation Through Structure”. In: *In Proc. of the ICNN-96*. IEEE, 1996, pp. 347–352.

[17] D. Grune and C. Jacobs. *Parsing Techniques: A Practical Guide*. Monographs in Computer Science. Springer New York, 2007. ISBN: 9780387689548.

[18] C. Guo et al. “On calibration of modern neural networks”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 1321–1330.

[19] H. Gurulingappa et al. “Development of a benchmark corpus to support the automatic extraction of drug-related adverse effects from medical case reports”. In: *Journal of biomedical informatics* 45.5 (2012), pp. 885–892.

[20] I. Hendrickx et al. “SemEval-2010 Task 8: Multi-Way Classification of Semantic Relations Between Pairs of Nominals”. In: *CoRR* abs/1911.10422 (2019). arXiv: 1911.10422. URL: http://arxiv.org/abs/1911.10422.

[21] N. Jadallah. *Cause-Effect Detection for Software Requirements Based on Token Classification with BERT*. Feb. 2021. URL: https://colab.research.google.com/drive/14V9Ooy3aNPsRfTK88krwsereia8cfSPc.

[22] A. Joshi et al. “The Penn Discourse Treebank 2.0 Annotation Manual”. In: *The PDTB Research Group* (2007).

[23] C. S. Khoo et al. “Automatic extraction of cause-effect information from newspaper text without knowledge-based inferencing”. In: *Literary and Linguistic Computing* 13.4 (1998), pp. 177–186.

[24] N. Kitaev and D. Klein. *Constituency Parsing with a Self-Attentive Encoder*. 2018. arXiv: 1805.01052 [cs.CL].

[25] N. Kitaev and D. Klein. “Tetra-Tagging: Word-Synchronous Parsing with Linear-Time Inference”. In: *CoRR* abs/1904.09745 (2019). arXiv: 1904.09745. URL: http://arxiv.org/abs/1904.09745.

[26] P. Kulkarni and Y. Joglekar. “Generating and analyzing test cases from software requirements using nlp and hadoop”. In: *International Journal of Current Engineering and Technology* 4.6 (2014), pp. 3934–3937.

[27] M. Kyriakakis et al. “Transfer Learning for Causal Sentence Detection”. In: *CoRR* abs/1906.07544 (2019). arXiv: 1906.07544. URL: http://arxiv.org/abs/1906.07544.

[28] F. Li et al. “A neural joint model for entity and relation extraction from biomedical text”. In: *BMC bioinformatics* 18.1 (2017), pp. 1–11.

[29] Z. Li et al. “Causality extraction based on self-attentive BiLSTM-CRF with transferred embeddings”. In: *Neurocomputing* 423 (Jan. 2021), pp. 207–219. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2020.08.078. URL: http://dx.doi.org/10.1016/j.neucom.2020.08.078.

[30] J. Liu and Y. Zhang. “In-order transition-based constituent parsing”. In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 413–424.

[31] Marcus, Mitchell P. et al. *Treebank-3*. 1999. DOI: 10.35111/GQ1X-J780. URL: https://catalog.ldc.upenn.edu/LDC99T42.

[32] F. Matthes. *Requirements Engineering (Software Engineering für betriebliche Anwendungen, Lecture 2, WS19/20)*. 2019.

[33] D. McClosky, E. Charniak, and M. Johnson. “Effective self-training for parsing”. In: June 2006, pp. 152–159. DOI: 10.3115/1220835.1220855.

[34] T. Mikolov et al. “Advances in Pre-Training Distributed Word Representations”. In: *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*. 2018.

[35] A. Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[36] J. Pennington, R. Socher, and C. D. Manning. “GloVe: Global Vectors for Word Representation”. In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[37] M. E. Peters et al. *Deep contextualized word representations*. 2018. arXiv: 1802.05365 [cs.CL].

[38] S. Pyysalo et al. “BioInfer: a corpus for information extraction in the biomedical domain”. In: *BMC bioinformatics* 8.1 (2007), pp. 1–24.

[39] K. Sagae and A. Lavie. “A classifier-based parser with linear run-time complexity”. In: *Proceedings of the Ninth International Workshop on Parsing Technology*. 2005, pp. 125–132.

[40] R. Socher, C. D. Manning, and A. Y. Ng. “Learning continuous phrase representations and syntactic parsing with recursive neural networks”. In: *Proceedings of the NIPS-2010 deep learning and unsupervised feature learning workshop*. Vol. 2010. 2010, pp. 1–9.

[41] R. Socher et al. “Parsing natural scenes and natural language with recursive neural networks”. In: *ICML*. 2011.

[42] R. Socher et al. “Semi-supervised recursive autoencoders for predicting sentiment distributions”. In: *Proceedings of the 2011 conference on empirical methods in natural language processing*. 2011, pp. 151–161.

[43] I. Sommerville. *Software engineering*. 8th ed. Pearson, 2007.

[44] M. Stern, J. Andreas, and D. Klein. “A Minimal Span-Based Neural Constituency Parser”. In: *CoRR* abs/1705.03919 (2017). arXiv: 1705.03919. URL: http://arxiv.org/abs/1705.03919.

[45] I. Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: *International conference on machine learning*. PMLR. 2013, pp. 1139–1147.

[46] A. Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[47] T. Watanabe and E. Sumita. “Transition-based neural constituent parsing”. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 2015, pp. 1169–1179.

[48] *What is Requirements Engineering? - Definition from Techopedia*. Apr. 2015. URL: https://www.techopedia.com/definition/21697/requirements-engineering.

[49] *Why Use Python for AI and Machine Learning?* Apr. 2021. URL: https://steelkiwi.com/blog/python-for-ai-and-machine-learning/ (visited on 07/18/2021).

[50] Y. Wu et al. “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”. In: *CoRR* abs/1609.08144 (2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.

[51] J. Yang, S. C. Han, and J. Poon. “A survey on extraction of causal relations from natural language text”. In: *arXiv preprint arXiv:2101.06426* (2021).

[52] M. Zhang. “A survey of syntactic-semantic parsing based on constituent and dependency structures”. In: *Science China Technological Sciences* (2020), pp. 1–23.

[53] Y. Zhang et al. “Position-aware attention and supervised data improve slot filling”. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 2017, pp. 35–45.

[54] M. Zhu et al. “Fast and accurate shift-reduce constituent parsing”. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2013, pp. 434–443.