On the Applicability of Deep Learning to Construct Process Models from Natural Text

Devos Simeon and Rogge Niels

May 13, 2019

Abstract

Business processes are a core asset in any organization. Constructing a conceptual process model is key in order to understand, analyze, improve and optimize a process. However, as most organizations document their processes in an unstructured textual format, constructing conceptual models can be a very time-consuming task. Algorithms that construct process models automatically from natural text could lighten the workload of a process modeler. As of now, such algorithms were entirely rule-based, an approach which does not generalize or scale well. Since most current natural language processing tasks rely on deep learning, we present a novel approach that leverages state-of-the-art deep learning models to transform a written text to a process model. Furthermore, an XML format to represent process models is proposed. This format allows both the training of seq2seq models as well as a smooth mapping to a graph. Considering sequences of activities and two types of XOR splits, we report a reasonable performance on a test set of 60 textual descriptions. All of our work as well as data is made publicly available on our Github. A Python implementation of our methodology is also available in a library called Description2Process.

Contents

1 Introduction

Motivation Business processes define the shape of any organization. They are the collections of all activities, jobs and responsibilities performed in an organization. Due to growing demands for globalization, innovation, and operational efficiency, organizations nowadays heavily rely on business process management (BPM) to analyze and optimize their business processes. One of the main concepts in BPM is the use of conceptual models, which visualize the different steps taken in a process (as well as the actors and resources involved). This not only allows to understand business processes but also to identify opportunities for improvement and eventually to optimize them. However, most organizations document their processes in texts written in natural language. (Blumberg and Aktre, 2003) estimated that more than 85 percent of all business information exists as unstructured data, which effectively means written natural language. Therefore, creating such conceptual models can be a very time-consuming task as the modeler needs to read all these documents and solve understanding issues through interviews, meetings, or workshops. Therefore, the automation of such conceptual models from natural language texts would be highly beneficial.

Why deep learning The field which is concerned with enabling computers to understand and process human languages is called natural language processing (NLP). Most state-of-the-art NLP techniques rely on machine learning and more specifically deep learning. The great benefit of deep learning is that it replaces the very time-consuming task of hand-designed feature engineering by learning the features itself from data (this is referred to as representation learning). At the same time, deep learning algorithms are able to achieve a higher accuracy than the techniques which rely on feature engineering on a variety of tasks. Therefore, it is interesting to see how the task of automated model generation could benefit from this.

Structure of the paper In section 2 we briefly review the existing related work as regards automated model generation from text. Next, in section 3, we provide the most important background in the fields of business process management (BPM), deep learning and natural language processing (NLP) one needs in order to understand the algorithms used in section 4. The latter forms the core of our paper, as it outlines our proposed approach to transform a text written in natural language to a process model. The data used as well as the results of our methodology are discussed in section 5. A discussion of the limitations of our approach and a look into the future are provided in section 6. We end our paper with a conclusion in section 7.

2 Related work

For an extensive overview as well as comparison of the approaches that have been followed to construct a process model from natural language text, we rely on (Riefer et al., 2016) and (Thom et al., 2018). The thesis of (Friedrich et al., 2011) is considered state-of-the-art and only one more but incomplete approach has been developed since then (Epure et al., 2015). We discuss the similarities and differences between both approaches below.

Similarities Both approaches are entirely rule-based. The common thread running through both is that first, parsers and taggers (as will be explained in 3.3) are used to analyze the textual description on a syntactic level. Besides a syntactic analysis, also a semantic analysis is conducted to extract the meaning of the words and phrases using available databases such as WordNet and FrameNet (see also 3.3). Next, all available information that results from the syntactic and semantic analyses is used to search for predefined templates and patterns appearing in the textual description to identify model elements such as activities, events and actors. These patterns involve domain specific databases with all combinations of signal words and word forms to identify model elements. For example, (Friedrich et al., 2011) defines a collection of stop word lists such as conditional indicators and sequence indicators which are looked after in the text.

Differences Comparing both approaches, it is clear that (Friedrich et al., 2011) is the most complete one, as it imposes minor constraints on the textual input and it considers all BPMN elements such as activities, events, data objects, pools and lanes. As part of the semantic analysis, the approach also developed an own anaphora resolution algorithm to resolve anaphoric references (again, more on this in 3.3). The approach of (Friedrich et al., 2011) is also much more complex due to a very extensive analysis: it consists of 29 algorithms in total which analyze a text both on a sentence as well as a text level. The approach of (Epure et al., 2015) on the other hand only consists of 2 algorithms: one to identify the activities and one to determine the relationships between the identified activities. The text is also only analyzed sentence by sentence. There is no explicit BPMN model generated, only a textual representation is returned (e.g. Start $\rightarrow$ (Act1 || act2) $\rightarrow$ Act4 $\rightarrow$ Stop). In terms of evaluation, (Friedrich et al., 2011) developed an own evaluation metric called similarity which is based on the Graph Edit Distance (GED) (Sanfeliu and Fu, 1983). It measures the similarity between a reference BPMN model and a generated one, ranging between 0 and 1. The authors report a similarity of 76% on a test dataset containing 47 model-text-pairs. (Epure et al., 2015) analyzed their proposed technique on a practical case, namely a methodology text of an archaeological project. They report a score of 88% correctly discovered activities in the text and a textual representation that adequately reflected the original process.

The main shortcoming of both approaches is that they are rule-based. Working rule-based using predefined patterns does not generalize or scale well, as every process description can describe a business process in a different way. Another shortcoming of rule-based approaches is that it leads to very complex algorithms, which require a lot of knowledge on the domain as well as a lot of manual work. Rule-based approaches are also not easily portable to another language: one typically has to start over, since rules are very language-specific.

The only field in which machine learning is applied in the existing approaches is in the parsers and taggers which are used to analyze the texts on a syntactic level. To this day, no approach has been developed that directly leverages machine learning to extract activities, events or other useful information regarding a process model from natural language texts.

The most important outcome of (Sintoris and Vergidis, 2017), which provide a summary of the state-of-the-art approach by (Friedrich et al., 2011), is that it is highly time for business process management to benefit from the latest insights and advances in the field of natural language processing. Hence, the motivation for this paper.

3 Preliminary background

Before discussing our methodology, we provide some background in the fields of business process management (BPM), deep learning and natural language processing (NLP).

3.1 Business process management

3.1.1 Goal of business process management

Business process management (BPM) is practiced within organizations to design, analyze, improve, optimize and monitor their business processes. As business processes represent a core asset of any organization, understanding them thoroughly is key in order to be able to exploit competitive advantages as well as to identify improvement opportunities. Improvement can be realised in different ways, depending on the objectives of the organization: reducing costs, cycle times and error rates are typical examples. Important to note is that BPM is not about improving the way individual activities are performed. Rather, it is about "managing entire chains of events, activities and decisions that ultimately add value to the customer and the organization" (Dumas et al., 2013). In order to make business processes as efficient and effective as possible, BPM relies on conceptual models using a modeling language such as BPMN to present them in a structured format.

3.1.2 BPM lifecycle

Within an organization, BPM takes a structured approach when it comes to improving business processes. The first step is process identification. One needs to determine which business processes need improvement, and for each of them, one or more appropriate performance measures (also called process performance metrics) should be identified. These performance measures will be used in the end to determine whether a process actually has improved or not. Such measures can be related to costs, cycle times and error rates. Once this has been done, the second step is to understand those business processes in detail: this is called the process discovery phase. It is in this phase that conceptual process models are created which reflect the "as-is" state of the processes. This phase is often the most time- and resource-consuming one of the entire BPM lifecycle (hence the motivation of this paper, see also 1). The third phase is process analysis, in which a thorough analysis of the processes is conducted to identify their issues. One also needs to determine, for each issue individually, what their impact is and what it entails to resolve them. After this, the process redesign or process improvement phase follows, in which one identifies how the issues can be resolved and how each process in its entirety can be improved (Dumas et al., 2013).

One typically relies on a number of redesign best practices (such as task elimination and resequencing of tasks) which are generally applicable. However, it is key to identify which heuristics suit the organization the best. It is in the process redesign phase that one moves from an "as-is" to a "to-be" state of the processes. Next, one needs to implement the changes in order to move to the "to-be" state. This phase is two-fold: it involves an overall organizational change management of all participants of the processes as well as process automation, which relates to how the underlying IT systems support the "to-be" state of the processes. The last (but certainly not least) phase of the BPM lifecycle is the monitoring and controlling phase, in which one needs to collect and analyze data of the processes to determine how well they are performing with respect to the defined performance measures. As new issues may arise within existing or new processes, the cycle must be repeated continuously throughout the existence of the organization (Dumas et al., 2013).

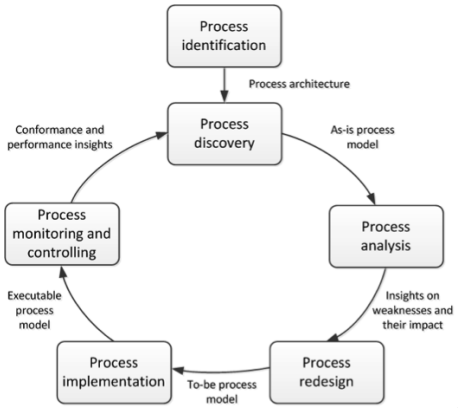


Figure 1: BPM lifecycle (Dumas et al., 2013).

3.1.3 Business process model notation

The Business Process Model and Notation (BPMN) is a standard graphical notation defined and managed by the Object Management Group[1]. The notation is used to create a visual overview of all the steps taken in a business process (in other words, a conceptual model). BPMN is one of the most widely used modeling languages to construct conceptual process models in the industry. It is supported by many modeling tools such as Signavio[2] and Bizagi Modeller[3].

The notation consists of four basic types of elements: events, activities, gateways and connecting objects.

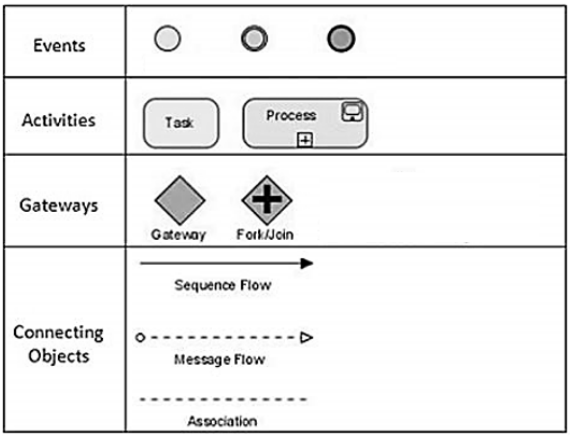


Figure 2: BPMN basic symbols[4].

Events are used to model something that happens instantaneously in a process. Events are passive elements: they just signal certain conditions or circumstances. BPMN makes a distinction between start events, intermediate events and end events. Start events indicate the beginning of a process, while end events indicate that a process instance is terminated. Anything that is signalled between the start and end of a process is modelled using an intermediate event. Common events include message events (which indicate the sending or arrival of a message) and timer events (which indicate that a certain deadline has been exceeded) (Dumas et al., 2013).

Activities capture units of work within a process. Contrary to events, activities are active elements: they are time-consuming and resource-demanding. BPMN makes a distinction between two types of activities: tasks and sub-processes. Tasks are atomic activities which are

[1] https://www.omg.org/bpmn/
[2] https://www.signavio.com/
[3] https://www.bizagi.com/en/products/bpm-suite/modeler
[4] https://study.com/cimages/multimages/16/3a2011dc-cdea-41e2-b6a2-ca25254e9f08_bpmn_symbols-2.png

not broken down further. On the contrary, sub-processes represent work that is broken down to a finer level of detail, being a process on its own. BPMN defines a variety of task types which can be particularly useful when modeling engineering requirements. However, tasks types are not much used in practice[5]. BPMN also provides the "Activity Loop" construct, which allows the repetition of a task or sub-process (Dumas et al., 2013).

Gateways can be used to indicate what can be done under which circumstances. They introduce splits (branches) and merges within the conceptual model. BPMN defines 3 types of gateways: exclusive (XOR), parallel (AND) and inclusive (OR). Exclusive gateways are the most commonly used ones, and represent an exclusive choice: only one path can be taken. An exclusive (also called simple) merge indicates that one can proceed when one branch has completed. Parallel gateways can be used to create and join parallel flows within a process. A parallel split indicates that all branches are taken, while a parallel merge indicates that one can only proceed when all incoming branches have completed. Finally, the (less frequently used) inclusive gateways can be used to include "one or many" logic in a process. With an inclusive split, one or several branches can be taken depending on the fulfilled conditions. An inclusive merge indicates that one can only proceed when all active incoming branches have completed (Dumas et al., 2013).

Connecting objects To connect all elements (among which events, activities and gateways) in a BPMN process model, 3 types of connectors are defined: sequence flows, message flows and associations. Sequence flows are used the most, since they allow to simply connect flow elements in a process. Message flows are used to synchronize different participants in a process (each having their own so-called pool[6]) with one another. One cannot use message flows to connect elements within the same pool. Associations are used to link information artifacts[7] with other BPMN elements (Dumas et al., 2013).

There is a lot more logic included in the BPMN standard not mentioned here which one can use to construct a conceptual process model. For an extensive explanation, we refer to (Dumas et al., 2013).

3.2 Deep learning

3.2.1 What is deep learning?

Deep learning is a specific kind of machine learning used to perform difficult tasks which cannot be handled by linear models, such as speech recognition, machine translation and object detection. Deep learning is characterized by the use of neural networks, which are loosely inspired by the understanding of the biology of our brains. Neural networks come in a variety of architectures. In the following section, we start explaining feedforward neural networks, which can be considered the most commonly used architecture. Later, we will discuss recurrent neural networks. Finally, we discuss sequence-to-sequence models (Goodfellow et al., 2016).

3.2.2 Feedforward neural networks

Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are the most basic deep learning models. The goal of a feedforward network is to approximate some function or concept f*. For example, for a classifier, $y = f(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation (Goodfellow et al., 2016).

[5] https://camunda.com/bpmn/reference/
[6] A BPMN pool represents a participant who takes part in a process.
[7] Artifacts represent data objects.

Some explanation about the name "feedforward neural networks":

Feedforward The reason these models are called feedforward is because information flows from the input all the way through the network to the output. There are no feedback connections in which the outputs of the model are fed back into itself. When this would be the case, they are called recurrent neural networks (RNNs), which are presented in section 3.2.3.

Networks These models are called networks because they consist of many different functions that are composed together. Each function is called a layer. The layers are arranged in a chain structure, with each layer being a function of the layer that preceded it. In case we have 3 functions (layers) $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$, and x is the input that is provided to the network, then the output of the last layer will be $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. The last layer in a neural network is called the output layer. The input vector x is called the input layer. The layers in between are called hidden layers. This is because the training data does not show what the output should be for each of these layers. The model itself must decide how to use those layers to produce the desired output.

Neural The architecture of neural networks is inspired by neuroscience. Each hidden layer is typically vector-valued: each element of the vector may be interpreted as playing a role analogous to a neuron in our brain. Rather than thinking of a layer as representing a single vector-to-vector function, we can also think of it as consisting of many units that act in parallel, each representing a vector-to-scalar function. Each unit in a layer is thus also referred to as a neuron.

Below, an example of a feedforward network is presented, drawn in two different styles. It consists of a single hidden layer h containing two units *h1* and *h2*. On the left, each unit (neuron) is drawn as a node in the graph. On the right, a more compact representation is shown, in which each layer is represented by a single node. The matrix W describes the mapping from the input vector x to h, and a vector w describes the mapping from h to y.

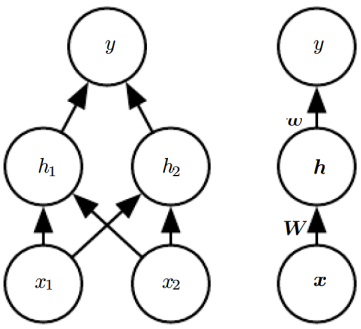


Figure 3: Example of a feedforward neural network, drawn in 2 different styles (Goodfellow et al., 2016).

The formulas for each layer are as follows (Goldberg, 2015):

- **input layer:** x is just a vector that contains the values for each of the n features.
- **hidden layer:** $h = f^{(1)}(x; W, c) = g(W^T * x + c)$.
- **output layer:** $y = f^{(2)}(h; w, b) = u(w^T * h + b)$

The c and b terms are called bias parameters. This terminology derives from the fact that the output of the transformations are biased toward being c and b respectively in the absence of any input. In the hidden layer, a nonlinear function g is applied to a linear transformation of x. In the output layer, a nonlinear function u is applied to a linear transformation of h. g and u are called **activation functions**. The choice for g and u is depicted below (Goodfellow et al., 2016).

Common non-linearities in the hidden layers The non-linear function g can take many forms. Below, the most common ones are discussed, based on (Goldberg, 2015).

ReLu The most used activation function is the rectified linear unit, abbreviated as ReLu. It is the default recommendation for use in the hidden layers of a feedforward neural network, since it delivers the best results, mainly because of the non-saturation of its gradient.

$$ReLU(x) = max(0, x) = \begin{cases} 0, & x < 0 \\ x, & otherwise \end{cases}$$

Hyperbolic tangent (tanh) The hyperbolic tangent tanh(x) is an S-shaped function, transforming the values x into the range $[-1, 1]$.

Non linearity in the output layer As for the non-linear activation function u, **softmax** is commonly used. This function can "squash" any K-dimensional vector of arbitrary real values to a K-dimensional vector of real values, where each entry is in the range (0, 1), and all the entries add up to 1. The output of the softmax function can thus be used to represent a categorical distribution – that is, a probability distribution over K different possible outcomes.

$$\mathbf{x} = x_1, ..., x_k$$
$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_i}}$$

Gradient-based learning Designing and training a neural network is not much different from training any other machine learning model: one needs to specify an objective function (which corresponds to minimizing the error on the training examples) as well as an algorithm for optimizing the objective function (Goodfellow et al., 2016).

- Objective function: the neural network, which is a parametric model, defines a distribution (denoted as p_{model}). To optimize the parameters, the principle of maximum likelihood is used (i.e. we choose as parameter estimate θ the one that maximizes the (log) likelihood function given the observed training data). Maximizing the log likelihood function is equivalent to minimizing the cross-entropy between the empirical distribution p_{data} and the model's predictions p_{model}. This is our objective function. The difference between the ground truth and the model's prediction is often called loss, and therefore the objective function is often referred to as the loss (or cost) function. Often, the total loss function will also incorporate a regularization term to avoid overfitting the training data (cfr. infra) (Goodfellow et al., 2016).
- Optimization algorithm: minimizing a loss function is done using gradient descent (i.e. in each iteration, you move a step in the direction of the negative gradient of the loss function). However, in practice one applies stochastic gradient descent (SGD), using minibatches of training examples, for performance reasons. The gradient itself is computed using the back-propagation algorithm (see further). When using a library such as Keras[8] or Tensorflow [9] for implementing deep learning models, one always has to specify an optimizer. All of them are extensions or variants of the classical stochastic gradient descent method. Famous variants include the Adagrad optimizer (Duchi et al., 2011) and the Adam optimizer (Kingma and Ba, 2014).

[8] https://keras.io/
[9] https://www.tensorflow.org/

Backpropagation Backpropagation (also abbreviated as **backprop**) is the algorithm that deep learning models use to efficiently compute the gradient, to be used in each (stochastic) gradient descent step. Actually, backpropagation is just the chain rule from calculus. One can calculate the partial derivative of the loss function with respect to a weight that is "deep" in the neural network by calculating the derivative of each tiny step all the way back to the weight, and then multiplying them altogether (Rohrer, 2017).

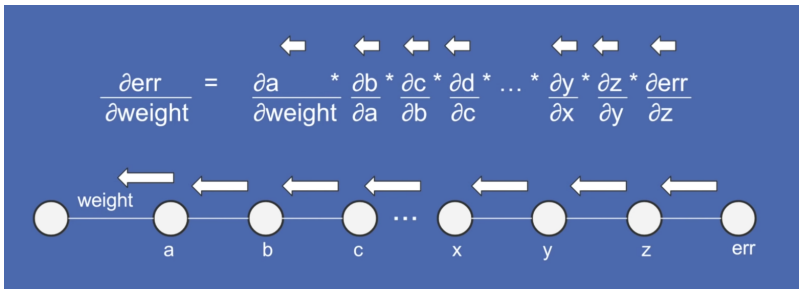


Figure 4: The idea of backpropagation (Rohrer, 2017)

More concretely, backpropagation is the algorithm used for determining how a single training example would like to adjust the weights and biases of the neural network. By calculating this for every individual training example and averaging over all examples, one gets a gradient descent step. In reality, one works in mini-batches for performance reasons as already mentioned, so one actually averages over all training examples in each mini-batch.

Generalization techniques In order to reduce/avoid overfitting the training dataset, some techniques are applied in order to increase the generalization of deep learning models. The most important ones are depicted below.

Weight decay: The weight decay approach is among the most popular regularization strategies. It consists of adding a term to the loss function that penalizes for weights being too big. This term is the L^2 norm (Goodfellow et al., 2016).

Dropout: Dropout is another very popular regularization technique for deep neural networks (Srivastava et al., 2014). Even the state-of-the-art models which have 95% accuracy can get a 2% accuracy boost just by adding dropout, which is a fairly substantial gain at that level. Dropout is fairly simple, and works as follows: when a training example is provided to the network, all input units and hidden units can be temporarily "dropped" or disabled with a probability p. The dropped-out neurons are resampled with probability p at every training step, so a dropped out neuron at one step can be active at the next one. Typically, an input unit is included with probability 0.8 and a hidden unit is included with probability 0.5. In that way, dropout samples from an exponential number of different "thinned" networks during training (Goodfellow et al., 2016).

Early stopping: Early stopping is a general technique used for generalization in machine learning, not just for deep neural networks. The idea is to stop training on the training dataset at the point when performance on a validation dataset (which is a separate dataset exactly used for generalization purposes that is not fed into the model for training) starts to degrade. This simple approach is very effective and widely used (Goodfellow et al., 2016).

3.2.3 Recurrent neural networks

When humans are reading a text document, they do not start comprehending every sentence from zero. Humans use the priorly read sentences and their context to comprehend the next sentence in the document. Feedforward neural networks cannot capture any notion of "history" or "memory" in them, which is a shortcoming in some cases. Recurrent neural networks (RNNs for short) address this issue by introducing loops in the neural network, allowing information to persist. The explanation below is based on (Goldberg, 2015) and (Olah, 2015).

The figure below visualizes the recursive architecture of a RNN. A simple neural network is presented with a box A. x_t represents the input at time step t and h_t is the output at time step t. The neural network looks at the input x_t and the previous output h_{t-1} to compute the output h_t. Next, h_t and x_{t+1} will then be fed into the same network to compute h_{t+1}. This process is repeated at each time step t. Unrolling the loop gives us a better understanding of how RNNs are able to process sequential data, as shown in the figure below.

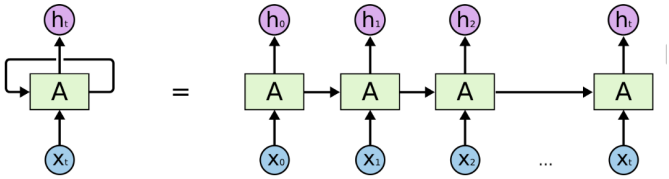


Figure 5: Graphical representation of a RNN. Left: recursive. Right: unrolled (Olah, 2015).

A standard (simple) RNN only uses a single layer within its neural network A. Mathematically, this can be written as follows:

$$s_t = A(s_{t-1}, x_t) = g(x_t * W^x + s_{t-1} * W^s + c)$$
$$h_t = s_t$$

$$x_t \in \mathbb{R}^{d_x}, s_t, h_t \in \mathbb{R}^{d_s}$$

The vector s_t is called the **hidden state**. Note that for a standard RNN, the output h_t at time step t is equal to the hidden state s_t at time step t. It is important to note that the same weight matrices and bias vector are used at each time step t. They are tuned during training with a variant of backpropagation suited for recurrent neural networks called **backpropagation through time** (Goodfellow et al., 2016).

Standard recurrent neural networks show great results in predicting the right output vector if the context needed in the prediction is close to the prediction itself. However, it turned out that in case the relevant information for a prediction is mentioned a couple of sentences before it is needed, RNNs are not able to learn such so-called "long-term dependencies". This problem was studied extensively by (Hochreiter, 1991) and is referred to as the vanishing gradient problem.

Luckily, long-short term memory networks and gated recurrent units have been developed specifically to address this issue.

3.2.3.1 Long-short term memory networks

Long-short term memory networks (LSTMs) offer the possibility to remember information for long periods (Hochreiter and Schmidhuber, 1997). In contrast to simple recurrent neural networks, the repeating neural network A in an LSTM has a more complicated structure. Instead of using a single layer, the structure of an LSTM contains four different layers interacting through pointwise operations. The figure below shows a graphical representation of the internal structure.

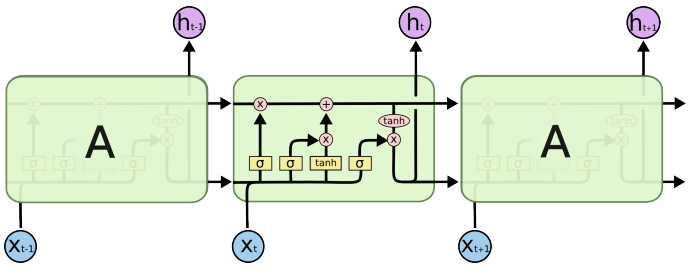


Figure 6: The repeating module in an LSTM contains four interacting layers (Olah, 2015).

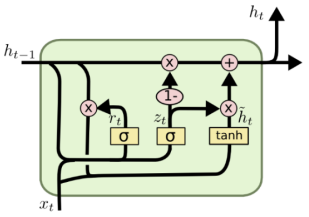
The crucial difference between standard RNN and LSTM neural networks is the addition of the **cell state** in addition to the hidden state. The cell state lets information flow along the chain of networks. It is only adapted by minor linear operations that can add additional information to it or remove outdated information from it. Mathematically, the equations are as follows:

$$
\begin{aligned}
i_t &= \sigma(x_t U^i + h_{t-1} W^i) \\
f_t &= \sigma(x_t U^f + h_{t-1} W^f) \\
o_t &= \sigma(x_t U^o + h_{t-1} W^o) \\
\tilde{C}_t &= \tanh(x_t U^g + h_{t-1} W^g) \\
C_t &= \sigma(f_t * C_{t-1} + i_t * \tilde{C}_t) \\
h_t &= \tanh(C_t) + o_t
\end{aligned}
$$

i, f and o are called the input, forget and output gates, respectively (Olah, 2015).

3.2.3.2 Gated recurrent units

Gated recurrent units (GRUs), introduced by (Cho et al., 2014), are a simpler variant of LSTMs: they combine the forget and input gates into a single "update gate". They also merge the cell state and hidden state. The resulting model is simpler than standard LSTM models, but its performance is comparable to that of an LSTM on sequence modeling tasks, with the additional advantage of having less parameters and being easier to train in certain circumstances (Olah, 2015).



$$
\begin{aligned}
z_t &= \sigma\left(W_z \cdot [h_{t-1}, x_t]\right) \\
r_t &= \sigma\left(W_r \cdot [h_{t-1}, x_t]\right) \\
\tilde{h}_t &= \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right) \\
h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t
\end{aligned}
$$

Figure 7: The repeating module in a GRU (Olah, 2015).

3.2.4 Seq2seq encoder - decoder network

Recurrent neural networks can be used to map an input sequence to an output sequence of the same length. However, many problems in NLP deal with input and output sequences of different lengths. For instance, the translation of an English sentence into a French sentence. The French translation is likely to not contain the same number of words as the input sentence. For such kinds of tasks, a different architecture called encoder-decoder was introduced (Sutskever et al., 2014; Cho et al., 2014). First, a RNN is used to encode the input sequence into a fixed vector representation (think of it as a "meaning" or "thought" vector), and this vector representation is then used as auxiliary input to another RNN that is used as a decoder to produce the output sequence. Such a model is also referred to as seq2seq since it takes a sequence (could be a sequence of words, characters...) as input and produces another sequence as output.

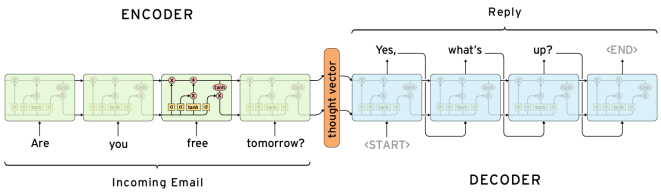


Figure 8: The encoder-decoder architecture[10].

Interesting to note here is that better performance is achieved when the encoder consists of 2 RNNs instead of just one, to encode the input sequence in 2 directions (from left to right and from right to left). This is referred to as a bidirectional RNN (Goldberg, 2015). The final vector representation of the input sequence is then a concatenation of the final hidden states of the 2 RNNS. For the decoder, a unidirectional RNN is typically used that goes from left to right.

3.2.4.1 The attention mechanism

However, it turned out that this basic architecture did not work well for longer sequences, since no matter the length of the input sequence, the encoder always turns it into a fixed-sized vector. Better results were obtained by using a new mechanism called attention, which was first introduced by (Bahdanau et al., 2014) and later refined by (Luong et al., 2015) and others. When we as humans translate a sentence, we pay attention to specific words which we are presently translating. This is the idea of the attention mechanism: instead of relying on just a single vector representation (encoding) for the entire input sequence, the encoding can be different at each time step of the decoding. In mathematical terms, one uses the current hidden state of the decoder $\mathrm{h_t}$ to score each of the hidden states $\bar{h}_\mathrm{s}$ of the encoder. These scores get normalized using a softmax to obtain the so-called attention weights $\mathrm{a_t}$. The weighted sum of the hidden states of the encoder is called a context vector $\mathrm{c_t}$. The context vector is then combined with the current hidden state of the decoder to yield the final attention vector α_t, which is used to derive the softmax logit and loss. The attention vector is then also fed as an input to the next time step to inform the model about past alignment decisions (this is called input feeding) (Luong et al., 2017).

$$
\begin{aligned}
\alpha_{ts} &= \frac{\exp\left(\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} && \text{[Attention weights]} \\
\boldsymbol{c}_t &= \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s && \text{[Context vector]} \\
\boldsymbol{a}_t &= f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_\boldsymbol{c}[\boldsymbol{c}_t; \boldsymbol{h}_t]) && \text{[Attention vector]}
\end{aligned}
$$

Figure 9: The attention mechanism (Luong et al., 2017).

3.2.5 Transformer

A novel neural network architecture for machine translation proposed by (Vaswani et al., 2017) called the Transformer really caused a stir in the NLP community. Instead of using complex recurrent or convolutional architectures, it entirely relies on feedforward neural networks and 2 advanced attention mechanisms called self-attention and multi-head attention in its encoder and decoder. Self-attention is the method the Transformer uses to bake the "understanding" of words in a sentence based on other relevant words in the same sentence. The reason it is called self-attention (or intra-attention) is that attention is applied within a language, as opposed to the aforementioned general attention mechanism. Multi-head attention on the other hand is the idea of computing multiple context vectors (heads) at each time step instead of just one. The intuition behind this is that different heads can focus on retrieving different types of information: one head could focus on the word to translate next, another head could provide extra context on the structure of the sentence, etc. (Vaswani et al., 2017).

Due to the fact that the model only relies on feedforward neural networks, it requires much less time to train as a lot of the calculations can be parallelized. It also turned out that the model generalizes well to other tasks like constituency parsing. For more information on its architecture, we refer to "The Annotated Transformer" by Harvard NLP (Klein et al., 2017).

[10] http://www.wildml.com/2016/04/deep-learning-for-chatbots-part-1-introduction/

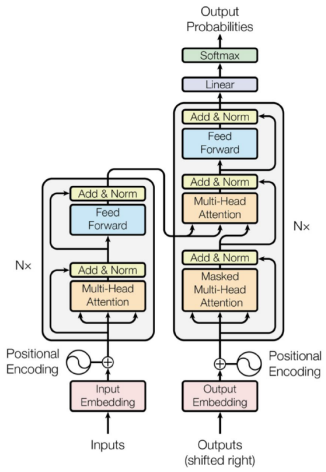


Figure 10: The Transformer model architecture (Vaswani et al., 2017).

3.3 Natural language processing

Natural language processing (NLP for short) is a subfield of Artificial Intelligence that is dedicated to enabling computers to understand and communicate in human language. The goal is to develop systems that improve the communication between humans and machines. The communication can be in written and/or spoken language. This paper focuses on written language. Statistical models are designed to understand the language structure or relations in a text to improve the communication (Chowdhury, 2003). A few examples of NLP applications include machine translation, text summarization, sentiment analysis and chatbots.

We first discuss word embeddings and the idea of transfer learning, as it is the core of current NLP. Next, we discuss the linguistic features extensively studied in NLP which we will use in our methodology. Each of the linguistic features are illustrated on a example sentence for the sake of understanding. For reporting on current state-of-the-art results, we rely on the NLP progress repository by Sebastian Ruder[11].

3.3.1 Word embeddings and the idea of transfer learning

As written natural language consists of words and characters whereas computers work with numbers, the words and characters need to be transformed into numbers in some way. Traditionally, NLP considers words as discrete atomic symbols, so they can be represented using a one-hot encoded vector that has as many dimensions as there are words in the vocabulary of the language. However, using this representation does not allow us to capture any semantic or syntactic "knowledge" about a word, nor does it allow to express any notion of similarity between 2 words (Mikolov et al., 2013).

The basic idea of **word embeddings** is to incorporate similarity between words within their vector representations. Words are represented (embedded) in a continuous vector space, such that words that are in some way similar to each other (such as "cats" and "dogs") have vectors that are close to each other in that space (Mikolov et al., 2013).

3.3.1.1 Pre-trained word embeddings

The first to come up with this idea was (Bengio et al., 2003). The main motivation for word embeddings was to reduce the dimensionality of the vectors from the size of the vocabulary (which can be 100,000) to a number between 50 and 1,000. However, the real popularization of word embeddings took place 10 years later when (Mikolov et al., 2013) created **Word2Vec**, which is a

[11] https://github.com/sebastianruder/NLP-progress

toolkit that allows both training and use of pre-trained embeddings. The embeddings are trained using a simple feedforward neural network (called Skip-Gram) that predicts the context given a target word[12]. The resulting vectors, which are the rows of the weights matrix between the input and hidden layer, can then be used to compute similarity. A commonly used similarity measure is the cosine similarity, which simply computes the cosine of the angle between the 2 vectors. Furthermore, it turned out that one can make clever calculations with the word embeddings. Take for example the following analogy: "man is to king like woman is to ...". We could compute this by solving the equation: vector('man') - vector('king') = vector('woman') - vector(answer). Then vector(answer) = vector('woman') - vector('man') + vector('king'). As it turned out, the answer gives (up to some approximation) the vector representation of the word 'queen'.

Word2Vec was followed by the release of GloVe (Pennington et al., 2014) and FastText (Bojanowski et al., 2016). Instead of deriving the word embeddings from a neural network, **GloVe** optimizes the embeddings directly so that the dot product of two word vectors equals the log of the number of times the two words occur near each other (within a specified window). **FastText**, open-sourced by Facebook in 2016, is an extension of the Skip-Gram model of Word2Vec, as it enriches word embeddings with subword information.

Pre-trained word embeddings (such as Word2Vec, GloVe and FastText) that are trained on large amounts of data such as the English Wikipedia corpus[13] can be downloaded from the web and directly used as first layer in any so-called "downstream" NLP task such as text classification. This layer is called the embedding layer. The pre-trained vectors can then either be treated as fixed during the network training process of the downstream task, or, more commonly, treated like any other parameters and further tuned to the task at hand (Goldberg, 2015). Furthermore, one can also just use an embedding layer as first layer in a neural network that learns the embedding vectors from scratch during training. This is only appropriate if enough data is available.

3.3.1.2 Contextualized word embeddings

Traditional word embeddings have an important limitation: they are context-independent. For every word in the vocabulary, exactly 1 vector is constructed. However, a word can have different meanings depending on the context in which it is used, think about the word "bank" for example. Recent advances in NLP ((McCann et al., 2017), (Peters et al., 2018) and (Akbik et al., 2018) were obtained by introducing contextualized embeddings, in which the embedding of a word depends on its context (i.e. the words left and right of it). In that way, a better understanding of the meaning of each word is captured. Here we discuss ElMo and Flair embeddings.

ElMo embeddings are derived from a pre-trained deep (2-layer) bidirectional LSTM language model[14], and are a function of the entire input sentence (Peters et al., 2018). The embedding of a word is a linear combination of all the hidden states of the pre-trained LSTMs, as well as an original word representation (for example, obtained from GloVe).

Flair embeddings are contextualized string embeddings, resulting from the hidden states of a bidirectional character-based language model using LSTMs (Akbik et al., 2018). The contextual string embedding of a word is then a concatenation of the output hidden state of the forward LM after the last character in the word, and the output hidden state of the backward LM before the word's first character. State-of-the-art results on tasks like NER[15] and POS tagging (see 3.3.3) are obtained by concatenating the Flair embeddings with traditional word-level embeddings such as GloVe.

[12]Word2Vec also has another variant which predicts the target word given the context. This architecture is known as Continuous Bag-of-Words or CBOW.

[13]https://www.sketchengine.eu/english-wikipedia-corpus/

[14]Language modeling is the task of predicting the next word or character in a document given the previous words or characters.

[15]Named Entity Recognition, the task of segmenting and classifying the named entities in a text, such as places and organizations.

3.3.1.3 Sentence embeddings

Instead of creating a separate vector for each word in a sentence, one can also directly embed an entire sentence or paragraph into a vector space. While simple baselines like averaging all word embeddings (also referred to as pooled embeddings) consistently give strong results, a few novel unsupervised and supervised approaches, as well as multi-task learning schemes, have emerged and lead to interesting improvements (Wolf, 2018). Here we only discuss the multi-task learning approach.

Universal sentence encoder embeddings (Cer et al., 2018) encode any body of text into a 512-dimensional vector. The model comes in 2 variants: one is based on the Transformer architecture (see section 3.2.5) and the other one is based on a Deep Averaging Network (DAN). The key finding of the paper is that the use of sentence embeddings tends to outperform the use of word embeddings in downstream NLP tasks.

3.3.1.4 Transfer learning

This idea of first pre-training a model on large amounts of data and then using (parts of) that model to conduct learning for a downstream NLP task (for which typically much less data is available) is called **transfer learning**. The embeddings discussed above are examples of this, as they result from a language model which was trained on large datasets. However, this form of transfer learning is typically referred to as feature-based as the embeddings are simply used as features in the first layer, whereas the downstream task still needs a very task-specific architecture. For example, if one wants to use those embeddings for text classification, one typically builds a feedforward neural network on top of the embedding layer, whereas for named-entity recognition, one uses a bidirectional LSTM with a conditional random field on top. A novel approach to transfer learning (the so-called fine-tuning approach) addresses this problem by introducing minimal task-specific parameters by simply fine-tuning the entire pre-trained model on the downstream task (Devlin et al., 2018). Here we discuss BERT.

BERT stands for Bidirectional Encoder Representations from Transformers, and is basically a Transformer (see 3.2.5) encoder stack trained on large unlabeled datasets (Devlin et al., 2018). BERT is trained on 2 tasks: as a so-called masked language model, in which the model needs to predict the original vocabulary id of a masked word in a sentence, and a sentence pair classification task, in which it needs to determine whether the second sentence follows the first sentence or is just a random sentence. BERT achieved state-of-the art on 11 NLP tasks, surpassing human performance on some.

The big advantage of BERT is that it can be used for a large variety of downstream NLP tasks by simply adding one additional output layer and then fine-tuning the pre-trained parameters. This is illustrated in the figure below.

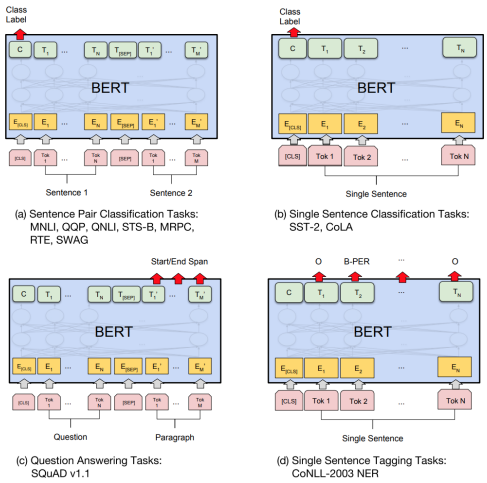


Figure 11: Incorporating BERT with one additional output layer (Devlin et al., 2018).

3.3.2 Tokenization

Tokenization is the task of splitting a text (could be a document, or a sentence) into meaningful segments, called tokens. One can tokenize a text into sentences, words, or even characters. Punctuation is also considered a separate token. In NLP, tokenization is often applied as a first step when processing a text. The tokens are then used as input in the next processing step.

Due to the fact that most tokens can be identified just by using the white-spacing characters in the text (in case of word tokenization), constructing a tokenization algorithm is actually not that difficult (at least for European languages). One starts by segmenting the text based on white-spacing characters, followed by iterating over the following 2 steps (Honnibal and Johnson, 2015):

- checking exception rules for each of the generated substrings. For example, in English, the substring "U.K." should remain one token, whereas punctuation at the end of a sentence should be split off.
- checking whether any prefix, suffix or infix can be split off from each of the substrings. This includes punctuation like commas, periods, hyphens and quotes.

As an example, applying word tokenization to the sentence "The claim is examined by a claims officer." results in 9 tokens:

[The], [claim], [is], [examined], [by], [a], [claims], [officer], [.]

3.3.3 Part-of-speech tagging

After tokenization, part-of-speech (POS) tagging can be applied to examine the syntactic category or **part-of-speech** of each token. Examples of syntactic categories are nouns, verbs, adjectives, conjunctions, pronouns, determiners and so on. A distinction is made between the open classes and closed clauses. The open clauses are part-of-speeches that occur in almost any language like nouns, verbs and adjectives. The closed classes are part-of-speeches that are language-specific, such as pronouns, conjunctions and determiners for the English language (Honnibal and Johnson, 2015). For a full list of universal POS tags, see[16].

The difficulty in POS tagging is that some words are ambiguous when it comes to to their part-of-speech. For instance, the sentence "I made her duck" can have multiple different meanings depending on how the word "duck" is used. Duck can refer to the noun, describing a bird or also the verb, crouch. A part-of-speech tagging algorithm should use the entire context around each word to determine its part-of-speech (Feferman, 2004).

Current state-of-the-art POS taggers are neural-network based and achieve an accuracy of nearly 98% on test data from the Wall Street Journal portion of the Penn Treebank. The architecture that is commonly used is a bidirectional LSTM with a conditional random field (CRF) on top (Ma and Hovy, 2016; Akbik et al., 2018).

Applying POS tagging to the aforementioned example sentence gives us the following:

token	**POS tag**
the	DET
claim	NOUN
is	VERB
examined	VERB
by	ADP
a	DET
claims	NOUN
officer	NOUN
.	PUNCT

[16] http://universaldependencies.org/u/pos/

3.3.4 Lemmatization

Another commonly applied task in NLP is lemmatization. Lemmatization is the task of assigning the base forms (so-called lemmas) of words. For example, the lemma of "walk", "walked", "walks" and "walking" is "walk", and the lemma of "cat" and "cats" is "cat". Lemmatization algorithms use part-of-speech-sensitive suffix rules and lookup tables (such as WordNet[17]) to transform each word into its lemma (Honnibal and Johnson, 2015). Applying lemmatization to our example sentence "the claim is examined by the claims officer" results in "is" converted to "be", "examined" to "examine" and "claims" to "claim". The rest of the words are already in their base forms.

3.3.5 Regular expressions

A regular expression (often abbreviated as "regex" or "regexp") is a sequence of characters that defines a search pattern that can be applied to text. Regular expressions have a standardised syntax that one has to use to define such a search pattern. An example of a regular expression can be the following:

Feb(ruary)? 23(rd)?

This regular expression defines a pattern that matches the strings February 23rd, February 23, Feb 23rd and Feb 23. The "?" character indicates that the previous character (in this case "ruary" and "rd") can occur zero or one times. The "?" character is only one of the so-called metacharacters which one can use to define a regular expression.

Regular expressions are useful not only to find certain patterns in text but also to perform "find and replace" operations to text (Goyvaerts, 2006).

3.3.6 Syntactic parsing

Syntactic parsing is the task of recognizing the syntactic structure of a sentence. There are 2 types of syntactic parsing, depending on how the syntactic structure is defined: constituency parsing and dependency parsing. In the former, the syntactic structure is defined by a context-free grammar, also called phrase-structure grammar. In the latter, the syntactic structure is defined by a dependency grammar (Jurafsky and Martin, 2018). We discuss both below.

3.3.6.1 Constituency parsing

Constituency parsing, also known as "phrase structure parsing", uses a context-free grammar (CFG) to recognize the sub-phrases ('constituents') in a sentence: noun phrases, verb phrases and clauses (Jurafsky and Martin, 2018). Especially extracting the clauses from a complex sentence will come in handy during our experiment.

A context-free grammar consists of a set of production rules, which are replacement rules. An example of such a production rule is S -> NP VP, which indicates that a sentence (S) can be split up into a noun phrase (NP) and a verb phrase (VP). However, a context-free grammar does not specify how the parse tree for a given sentence should be computed. We therefore need to specify an algorithm that employs this grammar to efficiently produce correct trees. The constituent parse for our example sentence is given in figure 12 below.

Recent approaches (Vinyals et al., 2015) convert the parse tree into a sequence following a depth-first traversal in order to be able to apply seq2seq models to it. The linearized version of the example parse tree looks as follows: (S (NP (DT) (NN)) (VP (VBZ) (VP (VBN) (PP (IN) (NP (DT) (NNS) (NN))))) (. .)).

As of the time of writing, the SOTA model achieving an F1-score of 95.13 on the Wall Street Journal section of the Penn Treebank applies self-attention (see 3.2.5) in the encoder and uses pre-trained ElMo embeddings (see 3.3.1.2) (Kitaev and Klein, 2018).

[17] https://wordnet.princeton.edu/

```
(S
  (NP (DT The) (NN claim))
  (VP (VBZ is)
    (VP (VBN examined)
      (PP (IN by)
        (NP (DT a) (NNS claims) (NN officer)))))
  (. .)))
```

Figure 12: Example of a constituent parsed sentence

3.3.6.2 Dependency parsing

Dependency parsing is based on the idea that linguistic units (words, or lemmas) are connected to each other by directed links, called dependency relations. Each dependency relation is a binary relation between a head and a dependent. The latter is also called modifier, since the modifier modifies the head (Jurafsky and Martin, 2018). A dependency parse of the example sentence is given below:

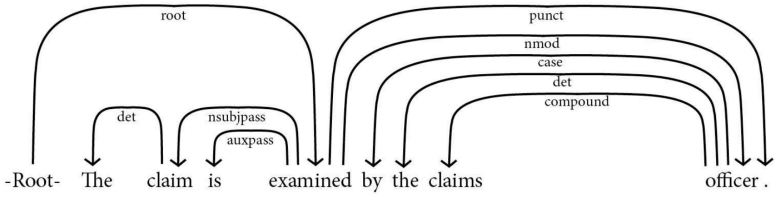


Figure 13: Example of a dependency parsed sentence

A list of 40 different dependency relations (labels) is defined by the Universal Dependencies framework[18]. The most important one is the 'root'. This refers to the main verb of the sentence, the head of the parse tree. Besides the root, a dependency parse allows to identify grammatical entities like the subject and direct object of a sentence. In a dependency tree, arrows always point from the head to the dependent. The dependency relationships directly encode important information that is often buried in the more complex constituent parses.

A major advantage of dependency grammars is their ability to deal with languages with free word order, such as Czech, Slovak and Turkish.

Dependency parsing algorithms can be divided into 2 types of approaches: graph-based and transition-based. Graph-based dependency parsing algorithms construct a graph of the sentence, in which each node represents a word and edges represent dependency relations between words. Current approaches treat graph-based dependency parsing as a classification task where the goal is to predict which in-going edge connects to each word x. Transition-based dependency parsing algorithms on the other hand build on shift-reduce parsing (Aho and Ullman, 1972) using an input buffer and stack. A neural network determines the appropriate transition operation to move words from the buffer to the stack or remove words from the stack. See (Chen and Manning, 2014) for more information on the basic idea.

Evaluation metrics for dependency parsing are the unlabeled attachment score (UAS) and the labeled attachment score (LAS). UAS does not consider the grammatical relation (e.g. nsubj) between the head and the child, whereas LAS requires a correct grammatical label for each attachment. Current state-of-the-art uses a graph-based approach and achieves a score of 96.61 and 95.02 on UAS and LAS respectively (Clark et al., 2018).

[18] http://universaldependencies.org/u/dep/

3.3.7 Semantic role labeling

Semantic Role Labeling (SRL for short) tries to recover the latent predicate argument structure of a sentence, providing answers to basic questions about sentence meaning, including "who" did "what" to "whom," etc. (Jurafsky and Martin, 2018). This task consists of 2 sequential steps:

1. Segmentation: one needs to determine which constituents in a sentence act as predicate or as semantic arguments to a predicate.
2. Classification: one needs to determine the semantic role that each of the identified arguments in the sentence play with respect to the predicate.

As for argument labels, different annotation schemes exist. The most used ones are PropBank and FrameNet. Both have corresponding resources with manually labeled sentences. As FrameNet's corpus is incomplete, we only discuss PropBank here. PropBank[19] (short for proposition bank) defines semantic roles in the form of numbers. In the general case, Arg0 indicates the proto-agent (the person or thing that initiates the action) and Arg1 indicates the proto-patient (the person or thing that undergoes the action). Semantics of other roles (indicated with Arg2, Arg3 and so on) are specific to verb senses. For example, the argument roles for the verb "examine" are defined as follows:

- Arg0: examiner
- Arg1: thing examined
- Arg2: in search of what

Following this definition, the sentence "the claim is examined by a claims officer" results in the following annotation: "$[\text{the claim}]_{\text{Arg1}}$ is examined by $[\text{a claims officer}]_{\text{Arg0}}$".

Current state-of-the-art semantic role labeling is achieved using a bidirectional LSTM-span model, achieving 87.4 F1 and 87.0 F1 on the CoNLL-2005 and 2012 datasets respectively (Ouchi et al., 2018). The model relies on ElMo embeddings (as explained in 3.3.1.2).

3.3.8 Coreference resolution

Coreference resolution is defined as the task of finding all expressions (noun phrases) that refer to the same entity in a text. These expressions are then grouped together in so-called entity clusters or chains (Jurafsky and Martin, 2018). Important to note here is that, contrary to all aforementioned NLP tasks, coreference resolution is typically applied on a discourse level, involving more than 1 sentence. Suppose that our example sentence is followed by a sentence that includes references to a previously mentioned entity:

"The claim is examined by a claims officer. In case it is not approved, it must be resubmitted."

This pair of sentences contains 2 entities ('the claim' and 'a claims officer') and 1 entity cluster: [the claim, it, it]. Coreference resolution is therefore two-fold: first, one needs to identify which spans in a text are entity mentions and second, one needs to determine, for each pair of entity mentions, whether they refer to the same entity. References are divided into anaphora, cataphora and exophora:

- anaphora: referring backwards (which is the case in the example above)
- cataphora: referring forwards e.g. "After he had received the invoice, the customer made a payment."
- exophora: referring to something extralinguistic, i.e. not in the same text e.g. "There are circumstances, where..."

At the time of writing, coreference resolution is far from being a solved task for computers. The current state-of-the-art, based on an end-to-end bidirectional LSTM model, achieves an average F1-score of ± 73% on the test set of the English CoNLL-2012 shared task (Lee et al., 2018).

[19] https://propbank.github.io/

4 Methodology

In this section, we outline our proposed methodology to transform a natural text description into a process model. In section 4.1, we discuss two approaches we considered to transform a textual process description into a process model using deep learning. The second one is the suggested approach and will be discussed in detail in section 4.2.

4.1 Approaches

4.1.1 End-to-end seq2seq model

The first approach consists of one deep learning model which is trained end-to-end: it gets a natural text description as input and outputs the corresponding process model in an XML-inspired structured format. This model is a sequence-to-sequence model (see sections 3.2.4 and 3.2.5) as it needs to learn how to map a sequence of words describing a process into a sequence of XML tags which eventually can be mapped to a graph.

Below, an example input and output of the end-to-end seq2seq model is given. The output can easily be transformed into a graph as shown in figure 14.

Input:

```
First a product is requested in the store. If it is a premium customer, the information is handled internally. Otherwise the order is confirmed. Then the shipping information is looked up in a database. When an invoice is sent, the payment is done by the customer.
```

Corresponding output:

```
<act> request product </act>
<path1>
    <act> handle information internally </act>
</path1>
<path2>
    <act> confirm order </act>
</path2>
<act> look up shipping information </act>
<act> send invoice </act>
<act> do payment </act>
```

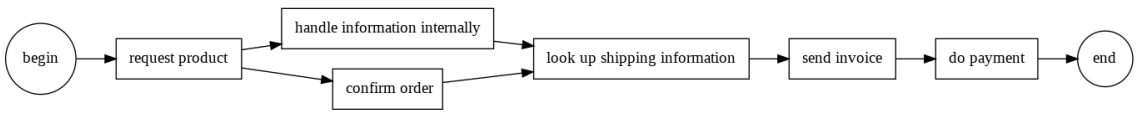


Figure 14: Visualization of the XML format

4.1.2 Pipeline

The second approach consists of an NLP pipeline. An NLP pipeline can be defined as a chain of independent modules, in which each module takes as input the output of the module that precedes it. A single module focuses on a subtask for the construction of a process model. Below, we briefly discuss the 8 sequential steps taken in the pipeline. Within each step, a dummy sentence is used as an example to clarify what has happened. The dummy sentence is the following:

> When the customer wants a gold card, they'll send it with the mail.

1. **Contraction expansion** First, we expand all English contractions in the textual description to their original form. For example, "doesn't" is expanded to "does not" and "he'll" to "he will". This cleaning step lowers the number of words in the vocabulary and thereby improves the training of deep learning models.

   > When the customer wants a gold card, **they will** send it with the mail.

2. **Coreference resolution** Next, all the references in the process description are resolved. Words like 'he', 'she' and 'it' lack meaning in the training steps and are also not expressive when they would be used in the final process model. Furthermore, activities can be repeated multiple times in a text. Coreference resolution helps to connect and recognize such repetitions as one activity (Riefer et al., 2016).

   > When the customer wants a gold card, they will send **a gold card** with the mail[a].
   >
   > [a] in order to resolve to what 'they' refers, one should have knowledge of the previous sentence.

3. **Clause extraction** Each sentence of the process description is split up into relevant shorter clauses. Further down the pipeline, it is easier to interpret the meaning of a shorter clause compared to longer complex sentences.

   > **clause1:** When the customer wants a gold card,
   > **clause2:** they will send a gold card with the mail.

4. **Activity recognition** Subsequently, a deep neural network (binary) classifies each of the obtained clauses into 'clause contains an activity[20]' or 'clause does not contain an activity'.

   > clause1: When the customer wants a gold card, → **no activity**
   > clause2: they will send a gold card with the mail. → **activity**

5. **Activity extraction** Next, for each 'activity' clause, an activity extraction algorithm searches for the short description of the activity in the clause. The short form of the activity is returned by the algorithm.

   > they will send a gold card with the mail. → **send gold card**

6. **Construct semi-structured description** In the next step, the original process description (cleaned during steps 1 and 2) is adapted to a semi-structured description. This is done by replacing the 'activity' clauses obtained in step 4 with the short form of their corresponding activity obtained in step 5, surrounded by XML tags. Conjunctions that describe the order of a process and splits/merges are not replaced during this step, as they will be used in the next step.

   > When the customer wants a gold card, <**act**> **send gold card** </**act**>.

[20] With activity, we mean a BPMN task, not a BPMN sub-process. See 3.1.3.

7. **From semi-structured description to XML format** A seq2seq model is used to translate the semi-structured description into a complete XML format of the process model. The main task of the model is to rightly order the activities and create correct splits and merges. The semi-structured description structurally represents key values to the seq2seq model for the creation of the final XML solution.

   ```
   <path1> <act> send gold card </act> </path1>.
   ```

8. **Model visualization** For human readability, the XML format of the business process model can be converted to a flowchart that is roughly based on BPMN.

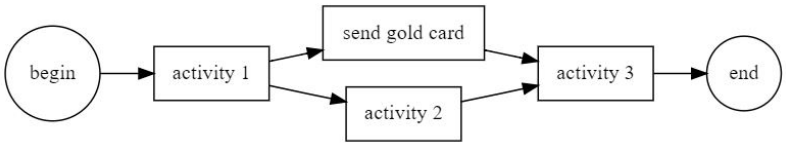


Figure 15: Example of process visualization

4.2 Pipeline in-depth

Below, we describe in detail our proposed pipeline. We created a Python library called "Description2Process" that collects and implements all the functionality of the pipeline. The code of this library can be found in a Github repository[21] and easily installed from PyPI[22]. Implementation details regarding this library can be found in the appendix, section A.1. The appendix also includes an example that shows the working of our methodology and the library (see A.2).

4.2.1 Contraction expansion

Textual process descriptions often include contractions such as "doesn't" and "isn't". In the first step of our pipeline, we expand these contractions for each sentence in order to diminish the length of the used vocabulary. More appearances of words like "not" in our training data will result in a better understanding of these words and their contexts by deep learning models further down in the pipeline.

Contraction expansion is not always as simple as you might think, as it sometimes requires contextual knowledge in order to choose the correct replacement words. Consider the following expansion rules:

```
he'd → he would
he'd → he had
```

If we encounter a sentence like "He'd like to know how he'd done that!", one needs contextual knowledge in order to know that the first expansion should be "He would" and the second one "he had".

A clever model exists to perform contraction expansion (Beaver, 2018). It considers all possible expanded versions of the given text, passes them through a grammar checker and sorts them according to least grammatical errors. In case the expanded version with least grammatical errors is unique, this version is returned. In case a tie exists between several expanded versions that have the same (least) number of grammatical errors, the model uses a distance metric called Word Mover's Distance (WMD) (Kusner et al., 2015) to measure the distance between the word embeddings of the original text and the word embeddings of each expanded version. Finally, the expanded version that has the smallest WMD is returned.

[21] https://github.com/NielsRogge/Description2Process
[22] www.pypi.org

4.2.2 Coreference resolution

In the data collection of textual descriptions (see 5.1), we encountered anaphoric sentences like:

> If the part is available, it is reserved. In case the part is not available, it is back ordered.

As the final process model should include the activities "reserve part" and "back order part", the computer needs to resolve the fact that "it" refers to "the part" twice. For this purpose, we made use of a pre-trained deep learning model that resolves coreferences[23]. The model is based on work by (Clark and Manning, 2016a,b) which achieves an average F1-score of ± 65% on the English CoNLL-2012 shared task, 8% lower than the current state-of-the-art (see section 3.3.8). The deep learning model is made of two sub-modules:

1. A rule-based mentions-detection module which uses POS tagging, dependency parsing and NER annotations to identify a set of potential coreference mentions.
2. A feedforward neural network which computes a coreference score for each pair of potential mentions.

4.2.3 Clause extraction

Sentences are often composed out of multiple clauses or subsentences. One of the main ideas of our pipeline is to reduce the complexity of the description by splitting it up into multiple individual clauses, as this will make the task of identifying and extracting the relevant activities for the final process model a lot easier.

The substructure of a sentence can be detected by performing a constituent parse of the sentence as explained in 3.3.6.1. We search for the constituents "SBAR" (which represents a clause that is introduced by a (possibly empty) subordinating conjunction) and "S" (which stands for simple declarative clause). "S" constituents are only searched under the prepositional phrase "PP", "S" itself, adverb phrase "ADVP", verb phrase "VP" and noun phrase "NP" constituents. Using this rule-based approach, we obtained reasonable results. Applying this to an example sentence encountered in our data is given below:

> **Sentence**: "The acceptance pack includes a repayment schedule which the customer needs to agree upon by sending the signed documents back to the loan provider.
>
> **Clauses**: ['The acceptance pack includes a repayment schedule], ['which the customer needs to agree upon'], ['by sending the signed documents back to the loan provider']

In this example, only the third clause contains an activity (namely "send signed documents"), so extracting activities on a clause level rather than the sentence level reduces the complexity.

It is important to note that we decided not to split up sentences that include the verbs "check", "decide", "determine" or "investigate" followed by "if", "which", "whether", "what" or where" because a sentence like "The department checks whether the master data can be changed at the desired time" would otherwise be split up into two different clauses "The department checks" and "whether the master data can be changed at the desired time", making it impossible to extract the correct activity "check whether the master data can be changed" on the clause level. This constraint has a rule-based implementation that uses lemmatization, as explained in 3.3.4.

[23] https://github.com/huggingface/neuralcoref

4.2.4 Activity recognition

Once the clause extraction algorithm has split up all sentences of the textual process description into separate clauses, a classification model needs to determine, for each clause individually, whether that clause contains an activity that should be included in the final process model or not. The ultimate goal of this supervised machine learning task is that the model learns which verbs in a clause are indications of an activity and which verbs are not. It is important to note that the context in which a verb appears is very important to determine whether it indicates an activity or not. Consider the following 2 clauses which we encountered in our data:

- "Next the customer is called on his phone (...)"
- "This is called the "three-way matching" process."

In the first clause, the verb "call" clearly indicates an activity: in the process model, we should include a task "call customer". However, in the second clause, the verb "call" is just used to give some more information on a particular process, a so-called meta-sentence. So it is clear that the model needs to take into account all words of the clause (or in other words, the context) in order to make its decision.

As this task is a text classification task, many different neural architectures exist in the literature, including convolutional (Kim, 2014; Kalchbrenner et al., 2014), recurrent (Tai et al., 2015; Wang et al., 2015) and recursive (Socher et al.) ones. However, convolutional architectures are very data heavy models (Young et al., 2017) and recurrent architectures such as LSTMs and GRUs are rather complex. As it turned out, simple feedforward neural networks work reasonably well and do not require a huge amount of data to train on. Due to their non-linearity, they have proven to report excellent accuracy on text classification tasks (Goldberg, 2015). As our labelled dataset is limited, we rely on the idea of transfer learning, as explained in 3.3.1.4. In that way, we leverage the insights from large pre-trained models for our supervised machine learning task. Below, we discuss both the feature-based transfer learning approach (by means of Word2Vec, ElMo, Flair and Universal Sentence Encoder embeddings) as well as the fine-tuning transfer learning approach (using BERT).

Word2Vec We used Word2Vec (see 3.3.1.1) embeddings as a baseline. The embedding layer is followed by a feedforward neural network consisting of 2 hidden layers having 10 nodes each. The embeddings are fine-tuned on the classification task.

ElMo and Flair The library that allows to use ElMo and Flair embeddings (see A.1) includes a text classifier that consists of just 1 linear layer with a softmax on top. As of now, it is not possible to customize this classifier[24]. However, one can experiment with different embedding combinations, including ElMo and Flair. The library supports both word embeddings and document (sentence) embeddings. Document embeddings can be obtained in 2 ways: either by calculating a pooling operation (such as the mean) over all word embeddings in the document, either by a RNN that takes the word embeddings of every token in the sentence as input and provides its last output state as document embedding. According to (Akbik et al., 2018), the best results are obtained using a concatenation of context independent embeddings (GloVe, FastText) and contextualized embeddings (ElMo/Flair). We considered all possible combinations.

Universal sentence Encoder As explained in 3.3.1.3, the Universal Sentence Encoder model comes in 2 versions: one trained with a deep averaging network (DAN) encoder, one with a Transformer encoder. The Transformer model is designed for higher accuracy, but the encoding requires more memory and computational time. The DAN model on the other hand is designed for speed and efficiency, sacrificing some accuracy. We used both as embedding layer followed by a feedforward neural network consisting of 2 layers having 10 nodes each. The embeddings are fine-tuned on the classification task.

BERT The idea is to fine-tune the entire BERT on the clause classification task. As BERT belongs to the fine-tuning category of transfer learning, minimal task specific parameters are introduced: one only needs to add 1 additional layer on top of the pre-trained model.

[24] https://github.com/zalandoresearch/flair/issues/604issuecomment-472200446

4.2.5 Activity extraction

Once the classifier has identified the clauses that contain an activity, the next step in our pipeline is to extract the activities from those clauses. For this, two approaches have been followed and compared. We outline them in detail below. Our final activity extraction algorithm combines both, as will be explained in 5.3.5.

4.2.5.1 Active/passive

The first approach is similar to that of (Friedrich et al., 2011): we first determine whether the clause is in the active or passive tense, and then based on that we extract the activities in the following rule-based manner (omitting some details):

- in case the clause is active, then activity = lemma of the root of the dependency parse + direct object
- in case the clause is passive, then activity = lemma of the root of the dependency parse + subject

To improve readability, we remove all determiners (such as 'a', 'the') and possessive pronouns (such as 'his', 'their') pronouns from the direct object/subject. This is also typically done when process models are created. Additionally, we check if the root verb has a child token with the "prt" dependency label (which stands for *phrasal verb particle* for verbs like "back UP", "fill IN", etc.). If this is the case, we include it in the activity. A clause like "the database is backed up" will then result in an activity "back up database" rather than just "back database".

For the determination of the active/passive tense, we wrote a function that simply searches for a token in the clause whose dependency label is 'auxpass' (which means, *auxiliary passive*) and whose lemma is 'be'. If such a token can be found, the clause is considered passive.

4.2.5.2 Semantic role labeling

As a second approach, we used semantic role labeling (SRL), as explained in section 3.3.7. Semantic role labeling suits the task of extracting activities from clauses quite well as it is capable of answering the "what" question with respect to a verb. For example, to extract the activity "submit purchase order" from the clause "the restaurant chain submits a purchase order", we need to answer the question: "Submit what?". The answer is most of the times given by the Arg1 argument label as defined by the PropBank annotation scheme, as this label is "usually assigned to the patient argument, i.e. the argument that undergoes the change of state or is being affected by the action"[25], in this case "a purchase order". An active/passive detector is not required here as an important goal of semantic role labeling (as outlined in the guidelines) is to "provide consistent argument labels across different syntactic realizations of the same verb", for example:

[The customer]$_{\text{Arg0}}$ receives [an invoice]$_{\text{Arg1}}$

[An invoice]$_{\text{Arg1}}$ is received

The model used is a deep bidirectional LSTM (He et al., 2017) and achieves an F1-score of 84.9% on the OntoNotes benchmark, which is quite reliable.

The basic idea is that the activity is composed of the lemma of the root verb of the clause followed by the ARG1 argument. Similar to the first approach, determiners and possessive pronouns are removed from the argument to improve readability. The 'prt" token is again included in the activity in case it is found. However, we had to include a number of things to this basic idea to deal with cases in which the root verb and/or ARG1 argument could not be found in the clause. These additions are outlined below.

- in case the root verb can be found but ARG1 not, use ARG2 as argument.

[25] http://clear.colorado.edu/compsem/documents/propbank_guidelines.pdf

- in case the root verb can be found but nor ARG1 nor ARG2 can be found, no argument is considered.
- in case the root verb cannot be found, check if ARG1 can be found. If this is the case, use the 'V" predicate that belongs to this ARG1 as root and ARG1 as argument.
- in case nor the root verb nor the ARG1 argument can be found, nothing is returned.

4.2.6 Construct semi-structured description

The next step in the pipeline is to recombine the original process description (cleaned during the contraction expansion and coreference resolution steps) with the extracted activities that were obtained in the previous step. The goal is to create a representation that is more straightforward to learn the final process model from compared to the original process description.

This representation is a semi-structured description which is built by replacing the clauses that were labeled as an activity in the activity recognition step by their corresponding activity that was found in the activity extraction step. Technically, these replacements are performed by means of regular expressions (see 3.3.5). Furthermore, the activity is already enclosed between the XML tags "<act>" and "</act>" when it replaces its corresponding clause. Consider an example below.

Original description: "The process starts when a product is requested in the store."

Clauses: ["The process starts when"], ["a product is requested in the store."]

Semi-structured description: "The process starts when <act> request product </act>."

For recognizing which part of the (cleaned) process description should be replaced, the algorithm tries out two major rules for each clause that has to be replaced. First, the algorithm checks if it can find an exact match between the clause and a part of the process description. Capital letters and all kinds of punctuation are not taken into account during the search.

Sometimes, an exact match can not be found because in the original description the beginning and the end of the clause are separated by another clause. In this case, the second rule is applied. This second rule looks at the first 10 and the last 10 characters of the clause that has to be replaced by its activity. If a match can be found for the first 10 characters in the description, the index of the first character of the match in the description is remembered as the starting point. A similar search is performed for the last 10 characters but here the last character is remembered as the ending point. If both indices could be found in the description, all characters between those two indices are replaced with the activity.

A list of predefined conjunctions, which will be introduced in 5.1.3, can be part of a clause that contains an activity. Such conjunctions that are part of an activity clause are not replaced with the structured activity. These conjunctions keep their original position in the description and will show their purpose in the next step of the pipeline.

The description is called semi-structured because it contains both structured and unstructured parts. Looking at the example above, the activity "<act> request product </act>" is written in a structured format. The clause which does not contain an activity, "The process starts when" is still written in its unstructured original form.

4.2.7 From semi-structured description to XML format

Once we have constructed a semi-structured description that contains the identified activities, the semi-structured description has to be transformed into a complete structured XML format of the process model. Therefore, two last critical tasks have to be carried out:

1. The activities need to be ordered in the order as described in the textual description of the process.
2. Splits and merges need to be correctly identified.

As we use an XML format for the final representation of the conceptual model, the two aforementioned tasks need to be captured using XML tags. Consider the following example to clarify how the translation from a semi-structured description to an XML format should happen.

Semi-structured description:

```
The process starts when <act> request product </act>. If it is a new customer, <act> ask
customer details </act>. Otherwise, <act> retrieve customer details </act>. Before <act>
send invoice </act>, <act> send package </act> . Then<act>check payment</act>.
```

XML format:

```
<act> request product </act>
<path1>
    <act> ask customer details </act>
</path1>
<path2>
    <act> retrieve customer details </act>
</path2>
<act> send package </act>
<act> send invoice </act>
<act>check payment</act>.
```

As one can see, <path> and </path> tags are introduced to represent splits and merges. This task of mapping a semi-structured description to an XML format can be seen as a sequence-to-sequence (seq2seq) problem. The deep learning architectures introduced in 3.2.4 and 3.2.5 are suited to solve such a task. In order for the seq2seq models to deal with the two jobs that were introduced at the beginning of this section, the models should be able to learn certain rules from the training data. Given our training data, we list three rules below that should be learned by a model in order to deal with the two tasks.

1. Include the activities enclosed between the XML tags "<act>" and "</act>" of the semi-structured description in the final XML format.
2. Use the conjunctions to place the activities in the right order.
3. Recognize conditional conjunctions to create correct splits and merges.

Rather than the existing approaches mentioned in 2, which use hand-crafted rules that search for certain conjunctions which indicate sequential (e.g. then, after) and conditional (e.g. if, whenever) flows in the text, the seq2seq model needs to learn to use these conjunctions in the semi-structured description itself to recognize how a particular process flow works.

We considered three different architectures: a vanilla seq2seq encoder-decoder network, a seq2seq encoder-decoder network that uses the attention mechanism, and the Transformer. These types of architectures are typically used to make a translation from one language into another language, for instance translate French to English. In our pipeline, the translation is made between the self-created language "semi-structured description" and the other self-created language "XML format of a process model". To obtain relevant results, these types of architectures are generally trained on hundreds of thousands or even millions of training examples (sentence pairs). Because of the limited available training data in our case, we decided to work as how traditional NLP systems

work: by encoding words as discrete atomic symbols (integers). Then we developed another data generation algorithm as will be described in section 5.3.7.1 that generates a large amount of pairs of (dummy) integer sequences that imitate integer-encoded semi-structured descriptions and their corresponding integer-encoded XML formats. In this way, the models can learn the three above mentioned rules from numbers rather than words.

Note that the three above mentioned rules can also be hard coded to make the translation from a semi-structured description to the XML format. However, this thesis investigates the applicability of deep learning for performing this task. If this type of deep learning models can learn these three rules on limited generated data, it is a possibility that the same approach can also be used to learn more complex rules from a substantial amount of semi-structured process descriptions. However, as mentioned before, such a substantial amount of training examples is unavailable to us at the time of writing. With even more training data, it could be possibility to replace the semi-structured descriptions with the original descriptions as proposed in the first approach.

The pairs of integer sequences generated in 5.3.7.1 are more or less random integers imitating integer-encoded semi-structured descriptions and corresponding integer-encoded XML formats. However, the words in the semi-structured descriptions and XML formats have a semantic meaning for a human-reader or modeller. Taking into account the meaning of the words, a person could for instance guess that an activity "make order" should happen before an activity"pay invoice". This semantic meaning is lost when using the integer data generation of section 5.3.7.1.

Once the model has learned the three basic rules from dummy pairs of integer sequences, the model can be trained further on real-life examples of semi-structured process descriptions and their XML formats. These descriptions and XML formats should then first be integer-encoded so that the encoder and decoder of the seq2seq model can read them as an input. When training on real-life semi-structured descriptions, the model can learn new rules about the order of the words as described in the previous paragraph.

Below, we discuss in detail three different seq2seq architectures that were considered during the research.

Vanilla seq2seq encoder-decoder network First, we worked with a vanilla seq2seq encoder-decoder model introduced in 3.2.4. This model serves as a baseline. The encoder consists of an LSTM layer with 256 nodes, with its input and output being both 200 sequential steps. The decoder does also consist of an LSTM layer with 256 nodes. Because the decoder takes the last output of the encoder as input, this is also 200 sequential steps. The output is limited to 130 sequential steps. This is set a bit lower because the output is typically way shorter than the input.

Note that practically not all descriptions have the same number of words, and that also goes for the dummy integer sequences. However, the input of this model takes a fixed number of sequential steps. Therefore, the sequences are padded with zeros until it reaches a length of 200. After the padding, the input is also one-hot encoded.

Seq2seq encoder-decoder network with attention Secondly, we studied a seq2seq encoder-decoder network, similar as the one above, but enriched with an attention mechanism, as described in 3.2.4.1. The pre-processing of the training data is done in a similar way as described for the vanilla seq2seq encoder-decoder network.

Transformer As described in 3.2.5, the Transformer is a novel architecture achieving state-of-the-art on a number of seq2seq tasks including machine translation. Google has developed and open-sourced a library called Tensor2Tensor (Vaswani et al., 2018) that allows everyone to use the original Transformer and various modifications of its architecture on a number of problems as well as own data.

As our datasets are limited (100,000 dummy pairs of integer sequences and 450 semi-structured descriptions with their XML formats), the architecture of the Transformer model (Transformer Base) were adapted to a tiny version of the original Transformer: 2 hidden layers, a hidden size of 256 and a filter size of 512.

4.2.8 Model visualization

As a final step, we visualize the generated process model by transforming the XML format into a graph. This step does not improve the quality of the model nor does it contribute to any of the deep learning models. This step is just included to increase the human readability of the business process model. For an example, see figure 8.

5 Empirical study

In section 5.1, we describe the data we collected to train the deep learning models of both the approaches introduced in 4.1. Next, in section 5.2, we describe the results obtained for the first approach, namely the end-to-end approach. In section 5.3, we report the results obtained for each of the steps of the second approach, namely the pipeline. Finally, in 5.4, we describe the overall performance of the latter, which is our proposed methodology.

5.1 Data

The training of machine learning models requires data. Deep learning models on average do require even more data to train on compared to other machine learning techniques like a regression model or a random forest (Xiong, 2017). Therefore, we consulted different sources to obtain the required data. In this section, we first list and discuss the acquired data. Thereafter, the issues with this data and the reason it could not be used to directly train a deep learning model that returns the conceptual model given its description is explained. Next, we demonstrate how the collected data is used to create data on which machine learning models can be trained effectively. Finally, we discuss which assumptions we had to make for the creation of our dataset.

5.1.1 Collected data

We collected data from a variety of sources. The data can be grouped into three broad categories:

1. Friedrich et al.: derived from (Friedrich et al., 2011). All data is publicly available on the Github page of Fabian Friedrich[26]. This includes data from a variety of sources such as Signavio and BPMN handbooks. However, the textual process descriptions sometimes lack their corresponding solution.
2. Sánchez et al.: derived from (Sànchez et al., 2018) and obtained after communication with Josep Sánchez from the Computer Science department of the Universitat Politècnica de Catalunya (UPC). Collection of 74 textual process descriptions together with their solution.
3. Other: the rest of the data was derived from Google using search terms such as "textual process descriptions" and "process modeling exercises", as well as the BPMN handbook used in the Business Process Management course at KU Leuven[27], which we both took. This collection mainly consists of exercises on BPMN.

Sadly, the BPM Academic Initiative Model by Signavio[28], which collects nearly 30,000 different BPMN models, turned out to be impractical due to various issues such as the absence of corresponding textual descriptions, models without activities in them and the SVG format.

An advantage of the fact that our data originates from a variety of sources is that the textual process descriptions vary a lot in terms of the kind of process they are describing (something we call a "**scenario**"). We have data on scenarios such as loan approvals and claim handling, but also on bicycle manufacturing as well as the workflow of hospitals and insurance companies, to name a few.

[26] https://github.com/FabianFriedrich/Text2Process/tree/master/TestData
[27] https://onderwijsaanbod.kuleuven.be/syllabi/n/D0I80AN.htmactivetab=plaatsen$_i n_h et_o nderwijsaanbod_i dp$1498048
[28] https://bpmai.org/download/index.html

5.1.2 Problem: Incomplete data

As described above, a nice collection of data was found. However, this collected data could not directly be used to train deep learning models. In most cases, either the data source only contained a description without a corresponding solution, either the solution was available without a corresponding description. Furthermore, solutions were often represented as an image of the BPMN model in JPG format. A more structured format for the solution (like XML or JSON) was required because learning from an image would have made things overcomplicated.

5.1.3 Solution: Data generation

Because of the lack of complete training data, we developed a data generation algorithm that can create process descriptions together with their corresponding solution in XML format based on the collected data. It is important to note that the original process descriptions are altered by the data generation algorithm to create a number of 'variants´ which are semantically similar.

Approach The approach can be seen as a data augmentation (bootstrapping) technique. The input of the data generation algorithm is based on 120 scenarios from our collected data. A scenario is a description of a certain setting in which certain activities occur. For each of the 120 scenarios, we split up the textual descriptions into subsentences and categorized them into 1 of 4 mutually exclusive categories, which are the building blocks for creating a process description. The categories are 'step', 'general info', 'condition' and 'condition result'[29]. All 4 categories are pretty straightforward: a 'step' subsentence describes a general activity in the process, a 'general info' subsentence just gives more information about the process without describing any activities, a 'condition' subsentence describes a condition to be fulfilled and a 'condition result' subsentence describes what should be done in case the corresponding condition is fulfilled. In case a subsentence contains an activity, the short description of the activity is attached to it as reference solution, surrounded by <act> and </act> tags. Below, an example subsentence is given for each of the four categories, together with their reference solution if applicable.

- **step** subsentence: 'the employee forwards the form to the Purchasing Department' <act> forward form </act>
- **general info** subsentence: 'collecting the signature from the supervisor takes on average five days'
- **condition** subsentence: 'in case the risk is found to be low'
- **condition result** subsentence: 'the credit is accepted' <act> accept credit </act>

All subsentences (and their reference solutions) were collected in one spreadsheet. The algorithm can then describe a given scenario in a variety of ways by randomly choosing between several possibilities. These are outlined below.

A first possibility on how the algorithm can alter a process description of a given scenario is by changing the conjunctions that are used to join the subsentences and create a coherent text. For instance, the two 'step' subsentences 'the customer pays for its order' and 'the pizza is baked' can be joined using different conjunctions without changing the semantic meaning:

After the customer pays for its order, the pizza is baked.

Before the pizza is baked, the customer pays for its order.

Similarly, a 'condition' subsentence can be described in another way, just by altering the conjunction:

If the customer is premium,

[29] Some minor details are omitted here for the sake of understanding.

In case the customer is premium,

For each obtained subsentence, we indicated whether or not it required a conjunction. The algorithm can then choose between different conjunctions to add to an individual subsentence or join two subsentences. Depending on the category of each subsentence, the algorithm can choose between different conjunctions. A complete list of the conjunctions that can be mutually changed is found in the appendix, section A.3. This list includes almost all conjunctions that were found in the collected data and new conjunctions can easily be added. Some subsentences do not require a conjunction from the list and have their own hard-coded conjunctions. An example could be "the invoice is then sent to the customer": the conjunction "then" is already included in the subsentence.

A second possibility in which the algorithm can alter a description is by changing the way a 'step', 'general info', 'condition' or 'condition result' subsentence is described itself. An example for a 'step' subsentence is given below.

A customer places a purchase order.

A purchase order is placed by the customer.

As one can see, one can say things differently by writing something in the passive rather than the active tense. However, there are other ways of writing things differently, such as using synonyms and writing things from a different perspective. An example for a 'step' subsentence is given below.

The customer receives feedback from the assessor.

The assessor provides feedback to the customer.

In the spreadsheet, we created multiple versions based on the aforementioned techniques for the same 'step', 'general info', 'condition' and 'condition result' subsentences. In that way, we made sure the algorithm could generate a large number of different semantically equivalent process descriptions.

Assumptions To create a coherent description, a conjunction is chosen randomly from the list of conjunctions or is hard-coded in the subsentences themselves. However, most subsentences within the spreadsheet require a predefined conjunction from the list. As a consequence, most descriptions generated by the algorithm use the same conjunctions. As this list is very similar to the conjunctions found in the collected data, the generated training data is representative for use by machine learning models.

For simplicity reasons, we made a number of assumptions regarding the descriptions that the data generation algorithm can create. One of them has already been mentioned: we only considered 4 types of building blocks to generate a process description: 'step', 'general info', 'condition' and 'condition result' subsentences. Furthermore, we assumed that each of the subsentences contain zero or one activity. This results in process descriptions in which sentences can contain at most two activities (since the data generation algorithm can join two 'step' subsentences using predefined conjunctions). This type of sentences occur most frequently in process descriptions, so this is an accurate assumption.

The 'condition' and 'condition result' subsentences can describe a split which should be included in the process model. Two types of splits are considered by our data generation algorithm:

Regular XOR[30] split with one activity in each branch before merging again. In this case, we included 'condition_path1' and 'condition_path2' subsentences in the spreadsheet with corresponding 'condition_result_path1' and 'condition_result_path2' subsentences that both contain an activity. The data generation algorithm creates the corresponding reference solution as follows: <path 1> <act> (activity of condition_result_path1) </act> </path1> <path2> <act> (activity of condition_result_path2) </act> </path 2>. We also included descriptions with a third condition (condition_path3) and corresponding condition result (condition_result_path3). The reference solution then also includes <path3> and </path3> tags.

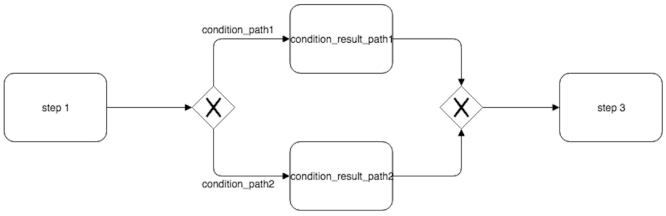


Figure 16: Regular XOR split.

XOR split with only one activity in a side branch before merging again. In this case, we included a 'condition_path1' subsentence in the spreadsheet with a corresponding 'condition_result_path1' subsentence that contains an activity. The data generation algorithm creates the corresponding reference solution as follows: <path 1> <act> (activity of condition_result_path1) </act> </path1>.

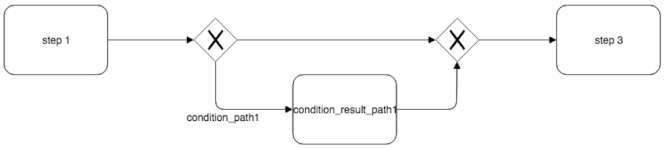


Figure 17: Regular XOR split with only one activity.

A last important assumption is the fact that for subsentences like "a message is sent" and "the person is notified", we included '<act> send message </act>" and '<act> notify person </act>" as reference solutions in the spreadsheet respectively. In BPMN, such subsentences would result in corresponding message events rather than activities. However, this paper focuses on sequences of activities and XOR gateways.

It is important to note that the training data of the deep learning models in this paper originates from the generated data. This means that these models actually reverse engineer what the data generation algorithm produces. However, the main building blocks are based on real process descriptions and the generated descriptions are very similar to the original ones in terms of semantics, except for the assumptions we made to simplify the solution. An advantage of this data generation algorithm is that its assumptions can easily be expanded in the future: one can add parallel and inclusive splits for example, or sentences containing more than two activities. More on this will be explained in section 6.2.

An illustration of the data generation algorithm on one particular scenario can be found in the appendix, section A.4.

5.2 Results of end-to-end approach

Despite the data generation algorithm, no reasonable results were obtained from any seq2seq architectures including the Transformer introduced in section 3.2.5. The output consisted out of

[30] see section 3.1.3

random activities that occurred in the training data but were not correlated at all with the given description. Hence, many more training data should be available in order to obtain reasonable results for this approach.

5.3 Results of pipeline steps

Based on the results of the end-to-end approach, we were motivated to construct a pipeline that sequentially focused on subtasks of the complete problem. Furthermore, the seq2seq models of the end-to-end approach are still used in the pipeline. However, the pipeline simplifies the input for the seq2seq models which allows them to learn to construct complete XML formats more conveniently. Below, we discuss the results obtained for each of the steps in our proposed pipeline.

5.3.1 Contraction expansion

The contraction expansion step is just a cleaning step to improve the training of the deep learning models further down the pipeline. We do not discuss any results here since in all of the textual descriptions of our collected data, no contractions were encountered with more than one possible replacement rule.

The contraction expansion that is included in the "Description2Process" library does not use the model discussed in 4.2.1 due to implementation issues. As a second-best approach, we used a dictionary of contractions together with their corresponding expansions. This dictionary is then used as a look-up table. We included this dictionary in the appendix, see A.5. However, it is important to note that the clever model is preferred over a simple look-up dictionary as it is still possible that process descriptions contain contractions with more than one replacement rule. In such a case, the clever model is superior over the dictionary as it uses learned word embeddings and a distance metric.

5.3.2 Coreference resolution

Our first idea was to simply apply coreference resolution using the deep learning model introduced in 3.3.8 at the level of an entire process description. However, this resulted in confusing and grammatically incorrect textual descriptions, as coreferent words were replaced many times by entire noun chunks in the wrong way. To limit this type of error, we decided to only resolve coreferences for the words "he", "she" and "it". Furthermore, we decided to perform coreference resolution on each sentence separately and only based on the previous (resolved) sentence. Using this sentence-pairwise approach, results were acceptable. To assess the performance formally, we set up a test set of 20 sentence-pairs[31] originating from our data generation algorithm that include coreferences which needed to be resolved. Of these 20 sentence-pairs, 14 of them (or 70%) were resolved correctly. A sentence-pair of our test set which was resolved correctly is given below. Coreferent words and their resolved versions are underlined.

Sentence-pair: "After receiving the sample, a physician of the lab validates its state. Subsequently he decides whether the sample can be used for analysis or whether it is contaminated."

Resolved version: "After receiving the sample, a physician of the lab validates its state. Subsequently a physician of the lab decides whether the sample can be used for analysis or whether the sample is contaminated."

A sentence-pair of our test set which was resolved incorrectly is given below. It is clear that the trained deep learning model lacks world knowledge to infer that something as "the request" can never make decisions.

[31] If a sentence is the first one in a textual description, no previous sentence was included, obviously.

Sentence-pair: "The IT department then sends the order to the supplier. Whenever management rejects the request, it can also decide to deny the request after which the process is ended."

Resolved version: "The IT department then sends the order to the supplier. Whenever management rejects the request, the request can also decide to deny the request after which the process is ended."

Despite the far-from-perfect performance of the model, we included it in our library since coreference resolution is an important task when transforming a textual process description into a conceptual model.

5.3.3 Clause extraction

Due to the use of an implementation of the state-of-the-art model for constituency parsing (see A.1), the performance of the rule-based clause extraction algorithm is quite well. We ran the clause extraction algorithm on a test set of 90 scenarios, with each having 3 semantically equivalent versions (so 270 textual descriptions in total generated by our data generation algorithm). As it turned out, only for longer and more complex sentence constructions involving multiple conjunctions such as "and" and "but", the clause extraction algorithm could output unexpected clauses not containing any verb. An example is given below.

Sentence: "It may be that the owner has already paid the fees with the request, in which case the cashier allocates a hearing date and the process completes."

Extracted clauses: ['It may be', 'That the owner has already paid the fees with the request', 'In which case and', 'The cashier allocates a hearing date', 'The process completes']

However, this is not a problem for the activity recognition step as such clauses will never be recognized as containing an activity.

5.3.4 Activity recognition

5.3.4.1 Training and evaluation data

In order to train and test the clause classification models introduced in 4.2.4, we needed examples (clauses) together with their label (activity/no activity). Of course, since we were the very first to use machine learning for this, we had to label the available data ourselves. We used the textual descriptions generated by our data generation algorithm on the 120 scenarios as explained in 5.1.3, with 3 semantically equivalent versions for each scenario, resulting in a total of 360 textual descriptions. We first applied our clause extraction algorithm to this collection of textual descriptions to obtain the individual clauses. To label them, our first idea was to compare the lemmas of the words of the reference solution in the spreadsheet to those of the clauses in order to label the clauses. However, as it turned out, this approach did not work well.

A better approach was to apply semi-supervised learning: we started off by manually labeling a number of clauses ourselves (a couple of hundreds), then we trained one of our classification algorithms on it, and then we used the trained classifier to label more examples, such that we only needed to correct its mistakes. This labeling approach was quite efficient.

As our data lacked clauses just describing the process on a meta-level (originating from the so-called 'general info' subsentences in the data generation algorithm), we decided to include some general descriptions about companies such that the model could better distinguish between clauses containing real activities and clauses just giving some general information.

For the creation of our training and test sets, we decided to use scenarios 50-70 of the 120 scenarios for testing and the rest for training. Next, we added the clauses originating from the

general company descriptions to the training and test sets using a 80-20 split respectively. For generalization, the training and test sets were shuffled. Finally, the training set was further divided into separate training and validation sets (again using a 80-20 split). Some summary statistics on our data can be found in the table below. Note that the classes (activity/no activity) are quite balanced. The reason our test set is 19% of the total data is because it is based on a number of scenarios rather than a number of clauses.

Table 1: Summary statistics on labeled data

	absolute	**relative**
total number of clauses	4,710	100%
number of clauses containing an activity	2,430	52%
number of clauses not containing an activity	2,280	48%
number of clauses from data generation algorithm	4,340	92%
number of clauses from general company descriptions	370	8%
number of clauses for training	3063	65%
number of clauses for validation	685	16%
number of clauses for testing	1,280	19%

5.3.4.2 Comparison

Firstly, we report the results for the Word2Vec, Universal Sentence Encoder and BERT models, as these all rely on Tensorflow. For Word2Vec and Universal Sentence Encoder, the Adagrad optimizer was used (see 3.2.2) with a learning rate of 0.003 and a (default) batch size of 128. For generalization, we used early stopping (as also explained in 3.2.2): we trained as long as the loss on the validation set did not decrease for 20 consecutive evaluations. Evaluations were performed every 1,000 training steps (i.e. training on 1,000 batches). We used a learning rate of 2e-5, a batch size of 32 and 10 training epochs. All performance metrics in the table below are reported over both classes on the test set.

Table 2: Comparison classifiers relying on Tensorflow library

	Precision	**Recall**	**F1-score**	**Accuracy**
Word2Vec	0.868	0.782	0.806	0.830
Universal Sentence Encoder - DAN	0.900	0.865	0.882	0.883
Universal Sentence Encoder - Transformer	**0.919**	**0.973**	**0.945**	**0.943**
BERT	0.918	0.935	0.927	0.925

These results confirm the insights from the literature: Word2Vec provides an acceptable baseline, but gets outperformed by the more state-of-the-art Universal Sentence Encoder and BERT models. The Universal Sentence Encoder model which uses a deep averaging network (DAN) scores substantially lower than the Universal Sentence Encoder model which uses the Transformer on all performance metrics. However, the former was much faster to train (20 minutes compared to 1 hour) on the same data. This also confirmed our expectations.

Models with ElMo and Flair embeddings were trained using a learning rate of 0.1, a batch size of 32 and a maximum number of epochs set to 150. The results are depicted in table 3 below. '+" signs in the first column indicate concatenations of embeddings. The FastText embeddings were trained over news and Wikipedia data. For Flair embeddings, we always used a combination of both forward and backward embeddings as recommended in (Akbik et al., 2018). For document pool embeddings, the mean was used as pooling operation. For document RNN embeddings, GRU (see 3.2.3.2) was used as RNN architecture. Document pool embeddings are immediately meaningful and are not fine-tuned on the classification task. However, document RNN embeddings need to be tuned on the downstream task. Precision and recall scores are reported for the "activity" class. For the F1-score and accuracy, we report the micro-averages over both classes. All performance metrics are reported on the test set.

Table 3: Comparison classifiers relying on Flair library

Model	**Precision**	**Recall**	**F1-score**	**Accuracy**
GloVe + ElMo document pool embeddings	0.906	0.865	0.887	0.796
GloVe + ElMo document RNN embeddings	**0.926**	0.874	0.901	0.820
GloVe + Flair document pool embeddings	0.869	0.849	0.859	0.754
GloVe + Flair document RNN embeddings	0.903	**0.903**	0.903	0.822
FastText + ElMo document pool embeddings	0.916	0.879	0.898	0.815
FastText + ElMo document RNN embeddings	0.915	0.894	0.905	0.826
FastText + Flair document pool embeddings	0.911	0.827	0.881	0.787
FastText + Flair document RNN embeddings	0.920	0.901	**0.910**	**0.836**

A number of things could be concluded from this table. First, it turned out that using fine-tuned RNN embeddings was in all cases (i.e. for every combination of pre-trained embeddings) superior to using pooled embeddings. Secondly, the best results in terms of F1-score and accuracy were obtained using a concatenation of FastText embeddings and Flair embeddings, as can be seen from the last row. However, comparing this to the results of table 2, this model is still outperformed by the Universal Sentence Encoder and BERT models on the same test set.

5.3.5 Activity extraction

To test and compare the performance of the 2 approaches introduced in 4.2.5, we created a test set of 400 clauses that contained an activity drawn randomly from the training dataset of 5.3.4.1. Next to each clause, we wrote the reference solution (activity) which needed to be extracted. For some clauses, it was appropriate to define multiple reference solutions, namely a shorter and a longer, more extended version. For example, for the clause "The information is registered into our production planning system", both "register information" and "register information into production planning system" were included as reference solutions.

As performance metric, we used the BLEU score (Papineni et al., 2002) which is a score for comparing a generated sentence to one or more reference sentences. A perfect match results in a score of 1.0, whereas a perfect mismatch results in a score of 0.0. It is one of the most widely used metrics for language generation problems such as machine translation and text summarization, due to its high correlation with human judgements of quality (Coughlin, 2003).

The BLEU scores[32] (averaged over the 400 clauses) can be found in table 4. As can be seen, both approaches performed reasonably well and extracted the reference solutions in most of the cases. However, for some verbs, the approach using semantic role labeling gave superior results compared to the simple active/passive rule as it is able to extract a more extensive explanation of the activity. This explains the higher BLEU score. A number of examples encountered in our test set that illustrate this are outlined below.

[32]Parameters used: 2 references for each clause, maximum n-gram length of 4, length penalty defined by the length of the shortest reference.

Clause 1: Under certain circumstances I must ask for corrections again
reference solution: ask for corrections
activity generated using active/passive: ask
activity generated using SRL: ask for corrections

Clause 2: We determine whether the customer is interested
reference solution: determine whether the customer is interested
activity generated using active/passive: determine
activity generated using SRL: determine whether the customer is interested

Clause 3: He informs the customer and the manager
reference solution: inform customer and manager
activity generated using active/passive: inform customer
activity generated using SRL: inform customer and manager

However, it turned out that were cases in which the semantic role labeling model did not work without apparent reason, returning an empty result. Therefore, we concluded that for our final activity extraction algorithm, we would prefer SRL over the simple active/passive rule, and only use the simple active/passive rule in case SRL did not return anything. In this way, the 2 approaches complement each other. Overall, this resulted in an algorithm that achieved a BLEU score of more than 90% on our test set. The combined approach is included in our library.

Table 4: Comparison activity extraction algorithms on 400 clauses

	BLEU
active/passive	0.8574
SRL	0.8911
combined	0.9076

5.3.6 Construct semi-structured description

As mentioned in section 4.2.6, the algorithm used for the creation of semi-structured descriptions is based on two major rules. Therefore, the performance of this algorithm does also fully depend on the performance of those two rules. When applying the first rule (exact match), only in one rare circumstance a mistake can be made, namely when the exact same clause does occur multiple times in the description and its first occurrence is part of a clause that should not be replaced. Otherwise, the first occurrence would have been properly replaced by its own activity and could not be found again during the search for the second occurrence. Under this rare circumstance, the wrong clause is (partly) replaced by an activity.

The second rule of this algorithm is way less robust than the first rule. However, this rule is only used when no exact match can be found. Find below a list of situations in which the second rule does not return the correct output.

1. It is possible that the first or the last 10 characters do not occur in a description because the separated part does only exist out of one or two words. More precisely, the part exists out of less than ten characters. As a result, the activity will not be put in the semi-structured description and thus also not occur in the final model.

   Sentence: The chef, who submitted it, receives a response with reasons.
   Clauses:

   - the chef receives a response with reasons
   - who submitted it

   First ten charachters: "the chef r"
   Last ten charachters: "th reasons"
   Replaced: Nothing (First ten characters can not be found)

2. It is also a possibility that the same ten characters already occur before the actual ten characters of the clause that is looked for in the description. In this case, the found index is too small and a too extensive part of the sentence is replaced. This type of situation does typically occur when the word expressing the subject of the description is quite long.

   Sentences: The system processes the device requirements. Device requirements are sent by the secretary, to the customer.

   Temporary semi-structured description:
   <act> process device requirements </act> device requirements are sent by the secretary, to the customer
   Clauses:

   - device requirements are sent to the customer
   - by the secretary

   First ten characters: "device req"
   Last ten characters: "e customer"
   Replaced: device requirements </act> device requirements are sent by the secretary, to the customer

3. In the case that a random clause A separates clause B that has to be replaced by the algorithm, clause A is positioned between the two detected indices from clause B. This type of scenario can result into two relevant situations. Clause A can be labeled as an activity or not. If clause A is labeled as an activity, this clause would be lost in the description and consequently also the activity. If clause A is not an activity, only some general info is lost. The loss of some general info will probably not have a high impact on the final model, for instance the two examples above. An example for when clause A is an activity is given below.

   Sentence: The news article which is always first approved by the editor-in-chief, is posted on the website.
   Clauses:

   - the news article is posted on the website
   - which is always first approved by the editor-in-chief

   First ten characters: "a news art"
   Last ten characters: "he website"
   Replaced: the news article which is always first approved by the editor-in-chief, is posted on the website

5.3.7 From semi-structured description to XML format

5.3.7.1 Training data

For training the three different models introduced in 4.2.7, we needed a collection of semi-structured descriptions together with their corresponding reference solution in XML format. For this, we first applied the data generation algorithm on 90 scenarios of our collected data, with five semantically equivalent descriptions for each scenario. This resulted in a collection of 450 process descriptions together with their reference solution in XML format. To transform the 450 process descriptions to semi-structured descriptions, we applied all steps prior to this step of our pipeline to each of the descriptions. The result is a training set of 450 semi-structured descriptions and their corresponding reference solution in XML format.

However, none of the three considered models were able to learn anything interesting from this limited training data.

Again confronted with the lack of training data for the deep learning models, a new specialized data generation algorithm was developed. Starting from the idea that the semi-structured

descriptions and XML formats would be integer-encoded, the new data generation algorithm was developed so that it would create a large amount of (dummy) pairs of integer sequences that resemble integer-encoded semi-structured descriptions and their corresponding integer-encoded XML formats.

The goal of the generated dummy pairs was to set up a test in order to study if certain deep learning architectures are able to learn the three rules described in section 4.2.7 from integers. If the test was successful, we could infer that these models can also recognize similar patterns in real (integer-encoded) semi-structured descriptions, under the condition that there is sufficient training data available. The obtained knowledge from training on integer dummy data can also be used for the initialization of a model that will be trained on real (integer-encoded) semi-structured descriptions.

Tokenizers are used to write a dictionary in which each word of the training data is allocated a unique integer. The first 50 integers are reserved for special tokens, namely XML tags and conjunctions. A list of the special tokens can be found in the appendix, section A.6. These tokens are called special because we assume that these tokens determine the structure of the final process model. Furthermore, there are around 1,300 other unique words encountered in the 450 semi-structured descriptions that have been assigned an integer.

The dummy integer data generation creates sequences of integers by combining the special tokens with the other 1,300 words that represent the content. The generation algorithm pays special attention to the use of the special tokens, so that they have the logical structure of a semi-structured process description. The content between special tokens, for instance between the two integers that represent the tags "<act>" and "</act>", is filled in randomly with the tokens that represent the content. Correspondingly, a sequence of integers that resembles the correct XML format is created, entirely rule-based. A more extensive example can be found in the appendix, section A.7.

Assumption It is important to note that the dummy integer data is highly focused around the special tokens. Descriptions that use conjunctions that are not included in the special tokens, are almost never seen by the algorithm. Therefore, it is likely the model will in this case not be able to return the right solution. However, including enough training examples with this missing type of conjunction should resolve this issue.

In total, 100,000 dummy pairs of integer sequences were created.

5.3.7.2 Comparison of the models on integer dummy data

As it turned out, only the Transformer model was able to learn the relevant structure from the dummy integer data. This can be explained by the complexity of the data: this type of data imitates a complete semi-structured description together with their XML formats, which are long sequences with a high number of different words and tags.

When experimenting on shorter sequences with less complex patterns, the encoder-decoder model with the attention layer performs significantly better compared to the model without the attention layer, consistent with the findings of (Bahdanau et al., 2014). The experiment used a sequence of 30 integers and the model had to return the integers between "1" and "2". "1" and "2" imitate the tags "<act>" and "</act>" and the integers in between resemble an activity.

As regards the Transformer model, the following hyperparameters were used: a learning rate of 0.05, a batch size of 1,000 and 40 epochs. The learning rate is varied over the course of training, according to the formula as described in the original paper (Vaswani et al., 2017). The results on the test set (using a 90-10 split for training and testing) are outlined in table 5 below.

Table 5: Performance of the Transformer model on integer data

accuracy	accuracy per sequence	approx. BLEU	neg. log pp.[33]	ROUGE-2 f1
0.995	0.800	0.813	-0.064	0.940

[33] negative log perplexity

The Transformer model that is currently included in the library to translate semi-structured descriptions to XML formats has been trained on 100,000 dummy pairs of integer sequences and uses beam search to make its predictions. The beam size is set to 4. This means that at each step the model keeps track of the 4 most likely continuations of the 4 partial translations of the previous time step. This leads to much better results than greedy decoding (Goldberg and Hirst, 2017).

5.4 Overall performance

In the previous section, we discussed the results for each of the components of our proposed pipeline separately. In this section, we assess the overall performance of the entire pipeline, i.e. we compare the reference solution of a description to the XML format generated by our pipeline. For this, we applied our "Description2Process" library on a test set of 20 scenarios (with 3 semantically equivalent versions for each scenario) created by our data generation algorithm as explained in 5.1.3. The 20 scenarios were not used to train any of the deep learning models which are present in our library. For clarification, the textual process description of one of the 20 scenarios, together with its reference solution, is given below.

Description: "In the process of adding a credit card, the user has to input his credit card details. Next, the billing address is checked. If it is incorrect, the user must update the address. Later the user needs to click the continue button to move forward. The credit card is then verified by the credit card authority. The result is then checked. If it is validated , the credit card is added. Whenever the result is not validated, the addition is cancelled."

Reference solution: <act> input credit card details </act> <act> check billing address </act> <path1><act> update address </act></path1> <act> click continue button </act> <act> verify credit card </act> <act> check result </act> <path1><act> add credit card </act></path1><path2><act> cancel addition </act></path2>

Applying the pipeline to this description resulted in the followed generated XML format:

Generated XML format: <act> add credit card </act> <act> input credit card details </act> <act> check billing address </act> <path1> <act> update address </act> </path1> <act> user needs continue button </act> <act> verify credit card </act> <act> check result </act> <path1> <act> add credit card </act> </path1> <path2> <act> cancel addition </act> </path2>

5.4.1 Metrics

To assess the performance formally, four different metrics are computed.

Number of activities The activity score is computed to evaluate the number of activities that are recognized by the pipeline. For both the reference solution and the pipeline's prediction, we count the number of activities in the XML format. Then the following formula is applied to compute the activity score:

$$n = \text{number of activities in reference solution}$$
$$m = \text{number of activities in pipeline's prediction}$$
$$score = \begin{cases} 1, & max(m,n) = 0 \\ 1 - \frac{|n-m|}{max(m,n)}, & otherwise \end{cases}$$

Number of branches The branch score is computed to evaluate the number of branches that are recognized by the pipeline. For both the reference solution and the pipeline's prediction, we count the number of paths in the XML format. More technically, we count the number

of times that a path tag (<path1>, <path2>, <path3>) occurs. The score is the computed with the formula below:

$$k = \text{number of branches in reference solution}$$
$$l = \text{number of branches in pipeline's prediction}$$
$$score = \begin{cases} 1, & max(k,l) = 0 \\ 1 - \frac{|k-l|}{max(k,l)}, & otherwise \end{cases}$$

BLEU score for order of tags To assess the order of the tags in a prediction of the pipeline, we use the BLEU metric as explained in 5.3.5. Here, we only focus on the tags, which means that we do not take into account the content between the tags. For both the reference solution and the pipeline's prediction, a sequence that only contains tags is deducted from the XML format. An example is given below. These sequences of tags are then compared with the BLEU metric that only uses bigrams.

XML format: <act> add credit card details </act> <act> check billing address </act> <path1> <act> update address </act> </path1> <path2> <act> click continue button </act> </path2> <act> verify credit card </act> <act> check result </act> <path1> <act> add credit card </act> </path1> <path2> <act> cancel addition </act> </path2>

Sequence of tags: <act> <act> <path1> <act> </path1> <path2> <act> </path2> <act> <path1> <act> </path1> <path2> <act> </path2>

BLEU score for order of activities To assess the order of the activities in a prediction of the pipeline, we again use the BLEU metric. Here, the focus is on the actual activities rather than on the tags. Before the BLEU score is computed, the individual words are replaced by a numbered token like "act1", "act2" and so on. In both the reference solution and the pipeline's prediction, the same activities get the same token. To determine whether two activities are the "same", we calculate a similarity score. If the similarity score is higher than 0.5, two activities are considered to be the same.

$$p = \text{number of words in activity in reference solution}$$
$$q = \text{number of words in activity in pipeline's prediction}$$
$$score = \begin{cases} 1, & max(p,q) = 0 \\ 1 - \frac{|p-q|}{max(p,q)}, & otherwise \end{cases}$$

Below, a short example is considered. In this example, the activities "add credit card details" and "add card details" have a similarity score higher than 0.5, so they both receive the "act1" token.

Prediction: add credit card details → check billing address → update billing address → verify credit card
Solution : add card details → check billing address → verify credit card

Prediction: act1 → act2 → act3 → act4
Solution : act1 → act2 → act4

A bigram BLEU score is computed based on the newly created sequences of tokens. This score evaluates the order in which activities occur, apart from the tags and splits.

5.4.2 Results

The scores for each of the four performance metrics on the test set can be found in table 6 below. As can be seen, the scores for "number of activities" and "number of branches" are relatively high. However, these two metrics only give an indication about the rough structure of the model generated by our pipeline. As these scores are around 80 percent, this means that the pipeline estimates the size of the model to be generated (based on the number of activities and branches) quite well.

As for the BLEU metrics, which assess the order in which tags and activities are generated, scores are also reasonable. In general, looking at the generated process models, the order of the activities is more or less right. This can also be explained by the chronological composition of a process description. Often, it happens that one activity is wrongly included or left out of the model, resulting in a lower BLEU in terms of activities. However, the Transformer model used in the final step does in most cases recognize the order quite well. For example, it is able to recognize the "before" pattern under which activities should switch position (e.g. "before the customer pays, the invoice is received" should become <act> receive invoice </act> <act> pay </act>).

Table 6: Overall performance of pipeline

Activities	**Branches**	**BLEU tags**	**BLEU activities**
0.853	0.755	0.702	0.562

Creating a process model from a process description without any mismatch between the solution and the prediction, is still quite uncommon. However, the errors made by the pipeline are often limited and can easily be fixed by a human modeller.

6 Discussion

6.1 Limitations

We discuss the limitations of our approach in terms of the type of textual process descriptions (in other words, the type of textual input) the pipeline can handle, the limitations regarding how a conceptual model is constructed based on a textual description (in other words, the model generation itself), and the limitations in terms of usability.

Limitations in terms of type of textual descriptions The assumptions of the data generation algorithm (outlined in 5.1.3) define the limitations regarding the type of textual input our approach considers: only descriptions containing sequences of activities and two types of XOR splits are considered. This leads to conceptual models containing only BPMN tasks, start and end events and XOR gateways (splits and merges). These elements are all connected by means of sequence flows. The current data generation algorithm does not create descriptions involving other BPMN constructs such as intermediate events, pools, message flows, information artifacts and sub-processes.

Limitations in terms of constructing a model As our approach consists of an NLP pipeline comprising a number of sequential steps, failures that happen in a particular step of the pipeline are propagated further down the pipeline:

- Failures in the **contraction expansion step** are almost non-existent, as pointed out in 5.3.1.
- Coreference resolution using neural networks is still a bit hit and miss (see 5.3.2), so failures in the **coreference resolution step** can replace noun phrases of a textual description in the wrong way, making it more difficult for the neural network to recognize activities, or resulting in inferior activities and/or splits and merges in the final XML format.
- The **clause extraction step** is rule-based, so it is possible that sentences are split at the wrong places, resulting in faulty clauses that serve as input to the activity recognition and activity extraction steps. However, as already pointed out in 5.3.3, the impact on the activity recognition step (and further steps in the pipeline) is limited.
- If a clause is not recognized correctly as a clause containing an activity in the **activity recognition step** (in other words, a false negative), the activity extraction step is simply skipped. This in turn results in a semi-structured description containing just the clause without the activity, and a final XML format which does not contain the correct activity. On the other hand, if a clause which does not contain an activity is incorrectly predicted to include an activity (in other words, a false positive), the activity extraction step is forced to search for a non-existing activity in the clause, which eventually ends up in the XML format. However, as performance of the activity recognition step is quite well (as pointed out in 5.3.4.2), such kind of failures are rather limited.
- The **activity extraction step** is responsible to extract the correct short form of activities to be included in the final generated solution. A faulty extraction immediately shows up in the final XML format.
- Failures in the **construction of the semi-structured description step** are possible as described in 5.3.6, so it can happen that clauses that contain an activity are not replaced correctly by their corresponding extracted activity.
- Failures in the final step, **going from a semi-structured description to an XML format**, can lead to activities being placed in the wrong order, or incorrect splits and merges. We consider this step the most volatile one, as it lacks training data despite the quite good performance on dummy integer data, see 5.3.7.

The current main limitation though of our pipeline is that the activity recognition step lacks global knowledge of the entire textual description. Currently, the deep learning model used in the activity recognition step must determine, for each clause individually, whether or not it contains an activity. In other words, activity recognition happens locally, on a clause level, rather than globally,

on the level of the entire textual description. This is not necessarily a bad thing, as searching for activities on a clause level rather than on a sentence or text level is much less complex. However, in reality, a human modeller uses the entire description to determine whether a clause contains an activity which should be included in the final process model or not.

It is perfectly possible that a clause appearing in some process description includes an activity that should be included in the final model, while a clause that consists of the same verbs occurring in another process description does not result in an activity. Take for example the clause "to determine the desired account type": does this clause contain an activity? In some process descriptions it does, in some it does not. It depends on the surrounding sentences. The current activity recognition algorithm does not use this context to determine whether or not this clause contains an activity, it simply has to make this decision solely on the words of this single clause (so it will label this clause as "containing an activity", as the verb "determine" most of the time indicates an activity). Also, due to this local classification, repeating the same activity in a textual description will result in two separate activities in the final XML format.

Limitations in terms of usability The purpose of the pipeline was not to create a fully functional program that can draw BPMN models from a process description. Rather, the pipeline does show that a combination of deep learning algorithms can be used to extract some fundamental information about a business process from text, such as knowing which verbs are indications of activities and the order in which these activities need to be placed. With more data and certain adaptations to the pipeline, the pipeline should be able to learn more complex BPMN elements. Automatically recognizing these BPMN fundamentals could speed-up the work of a human modeller.

6.2 Future work

In this thesis, elementary steps were taken to develop an approach that applied deep neural networks for the transformation of a natural text to a process model. The developed pipeline in this thesis does already show quite some functionality but it can still be extended largely. Below, we list some of our ideas to further improve the proposed pipeline.

Increasing the variety of the vocabulary Currently the training data of process descriptions is still quite limited. It consists out of 120 scenarios, which contain around 1,300 unique words. According to Education First, 90% of English conversations can be understood with 2,500 to 3,000 words[34]. All common English words should occur multiple times in order to have a complete, balanced training set. Furthermore, adding more process descriptions would also increase the variety in grammatical structures which need to be understood by the algorithms.

Adding BPMN elements The current textual inputs are limited to sequences of activities and XOR splits and merges as described in section 5.1.3 and 6.1. As a consequence, only these type of structures occur in the training data and can be generated as an output of the pipeline. However, as already pointed out earlier, the assumptions of the data generation algorithm can easily be extended to include descriptions that involve more complex BPMN constructs, provided that one introduces corresponding XML tags in the reference solutions. A number of examples are given below.

Adding other types of XOR splits One can include new types of XOR splits besides the ones described in this paper, such as a split with one branch in which the process is ending. In this case, one can include a 'condition_path1' subsentence in the input of the data generation algorithm with a corresponding 'condition_result_path1' subsentence with the latter not containing an activity. The data generation algorithm can then create the corresponding reference solution as follows: <path 1> <stop> </stop> </path1>.

[34] https://www.ef.com/wwen/english-resources/english-vocabulary/top-3000-words/

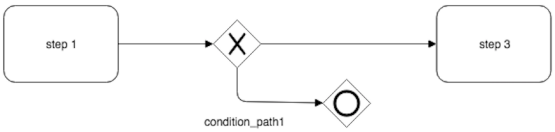


Figure 18: Regular XOR split with process ending.

Parallel gateways If one wants to add parallel (AND) splits and merges to the textual descriptions, one can add "<parallel>" and "</parallel>" tags to the reference solutions. One can also add conjunctions describing parallel splits and merges to the list of predefined conjunctions (such as "while", "meanwhile", "in the meantime") to let the data generation algorithm create a large variety of semantically equivalent descriptions containing parallel flows which the Transformer in step 4.2.7 can learn to understand.

Loops Loops often occur in process descriptions. Certain words or conjunctions like "back" and "again" point out the presence of a loop (Friedrich et al., 2011). These type of words or conjunctions can be added to the list of predefined conjunctions and eventually be recognized by the Transformer. XML also allows to add attributes to tags, so one can easily add "<act loop=True>" for example.

Events Adding events (like message and timer events) to the descriptions and XML formats is also a possibility. One can easily add "<event>" and "</event>" tags to the reference solutions. To indicate the type of event, one can add attributes, like "<event type=message>".

Actors and lanes A BPMN model does also show who is executing a certain activity. This person or system is called the actor and is indicated by a so-called swim lane in the model. An actor recognition algorithm similar to the activity recognition algorithm (4.2.4) could be developed to label which actor is executing which activity. The actor can then be represented in the solution by adding an attribute to the activity tag, for instance "<act actor='employee'>".

A Transformer for activity extraction The activity extraction step of our pipeline uses a combination of semantic role labeling and a simple active/passive rule. We believe that with more available data, it should be possible to train the Transformer seq2seq model to directly extract activities from clauses.

Adding a global module on top of activity recognition As outlined in section 6.1, the activity recognition step of the pipeline happens locally, on each clause individually. However, there is need for a global module on top of that that oversees all recognized activities and can detect repeated mentions of the same activity and remove incorrect or inappropriate activities.

Enhancing the 'Description2Process' library Currently, our library uses a simple look-up dictionary to expand contractions in the textual descriptions. The more advanced model for contraction expansion that relies on pre-trained embeddings, as outlined in section 4.2.1, still needs to be implemented.

7 Conclusion

In this paper, we presented a novel approach to automatically transform a text written in natural language to a process model. As this thesis inspects the applicability of deep learning on this task, an XML format for process models was introduced to allow the training of seq2seq models. We developed a data generation algorithm that can generate descriptions together with their solution in XML format based on the collected data, making some simplifying assumptions. However, despite this data generation algorithm, directly training a seq2seq model end-to-end was not possible due to a clear lack of data. Therefore, our proposed approach consists of a pipeline that splits up the complex problem into smaller subproblems that are more straightforward to learn. Still when solving particular subproblems of the pipeline with a deep neural network, we were often confronted with a lack of training data. Therefore, we had to cleverly transfer the knowledge from pre-trained deep learning models into the pipeline. The results of the individual components of the pipeline are good, especially for the clause extraction and activity recognition steps. However, the overall performance is lower due to the sequential execution of the components. Yet, the pipeline is capable of estimating the size of the model and the order of the activities quite well. Furthermore, attention was paid to the extensibility and improveability of the pipeline.

Combination of multiple deep learning models in order to tackle the lack of data. Rather than training a seq2seq model directly on a raw process description, the proposed pipeline performs six sequential steps on a raw process description to simplify the input for the seq2seq model: contraction expansion, coreference resolution, clause extraction, activity recognition, activity extraction and the construction of a semi-structured description. The sequential steps themselves leverage the knowledge from pre-trained deep learning models, as these steps themselves lack data: the coreference resolution step relies on a pre-trained feedforward neural network. For clause extraction, we rely on a pre-trained state-of-the-art deep learning model for constituency parsing. In the activity recognition step, we rely on pre-trained embeddings. In the activity extraction step, we rely on a pre-trained dependency parser as well as a pre-trained semantic role labeling model, which both rely on deep learning. The output of these six sequential steps, the semi-structured description, must then be transformed into a complete XML format by a seq2seq model. Still confronted with the lack of training data for this task, 100,000 integer sequences that resemble semi-structured descriptions, were generated. The insights captured by training the Transformer seq2seq model on these integer sequences could then also be applied on real semi-structured descriptions.

Attention for the extensibility and improveability Improvement of the included deep learning models will also improve the overall performance of the pipeline itself. Furthermore, the pipeline is designed in such a way that an individual step of the pipeline can be adapted without having an impact on the other steps. In this way, further research can also focus on a subtask of the problem. The proposed XML format for a process model can also easily be extended to include more elements from the business process model notation. Lastly, enhancing the data generation algorithm can lower the number of limitations that had to be made and increase the number of training examples.

The proposed pipeline is not constructed as a replacement for a human modeller. Moreover, in the current version, it will hardly simplify the modeller's work. However, the approach shows that deep learning is not only applicable when an abundance of training examples are available to find the demanded patterns. Cleverly combining pre-trained deep learning models does also deliver insights for the construction of a process model. Furthermore, the use of machine learning techniques shows advantage over the already existing rule-based approaches in terms of generalizability and the possibility to re-use the pipeline for different languages.

A Appendix

A.1 List of Python libraries

Below, we list the libraries that were used for the implementation of the pipeline.

Library	Usage
Pandas	Provide data structures and data analysis tools
Numpy	Support computational jobs
Spacy	Split up textual descriptions into individual sentences, perform dependency parsing, lemmatization and POS tagging
Pycontractions	Cleverly expand English contractions
Re	Python implementation of regular expressions
NeuralCoref	Coreference resolution
Benepar	Constituency parsing
NLTK	Create tree structure from syntactic parsed sentence, implementation of BLEU
Tensorflow	Construct deep learning classifiers with Word2Vec, Universal Sentence Encoder embeddings and fine-tune BERT
Flair	Implementation of a text classifier with FastText, Flair and Elmo embeddings
AllenNLP	Implementation of a deep biLSTM model for semantic role labeling
Keras	Construct LSTM encoder-decoder models
Tensor2Tensor	Implementation of the Transformer model
XML	Set-up navigational XML tree
Graphviz	Visualize process model

A.2 Example working 'Description2Process' library

Here, a short overview of the working of the 'Description2Process' library is given. The library consists of 10 modules which correspond to the 8 steps of the pipeline, a data generation module and lastly an evaluation module. For more information, we refer to our Github repository.

Data generation

```
description, solution = d2p.data_generation.get_description()
print(description)
print(solution)
```

First the secretary clarifies the shipment method. Whenever there's a requirement for large amounts, he selects one of three logistic companies. Whenever small amounts are required, a package label is written by the secretary. Subsequently the goods can be packaged by the warehousemen. If everything is ready, the packaged goods are prepared for being picked up by the logistic company.

<act> clarify shipment method </act> <path1><act> select one of three logistic companies </act></path1><path2><act> write package label </act></path2> <act> package goods </act> <path1><act> prepare packaged goods </act></path1>

Contraction expansion

```
description = d2p.contraction_expansion.expand_contractions(description)
print(description)
```

First the secretary clarifies the shipment method. Whenever **there is** a requirement for large amounts, he selects one of three logistic companies. Whenever small amounts are required, a package label is written by the secretary. Subsequently the goods can be packaged by the warehousemen. If everything is ready, the packaged goods are prepared for being picked up by the logistic company.

Coreference resolution

```
description = d2p.coreference_resolution.resolve_coreferences(description)
print(description)
```

First the secretary clarifies the shipment method. Whenever there is a requirement for large amounts, **the secretary** selects one of three logistic companies. Whenever small amounts are required, a package label is written by the secretary. Subsequently the goods can be packaged by the warehousemen. If everything is ready, the packaged goods are prepared for being picked up by the logistic company.

Clause extraction

```
clauses = d2p.clause_extraction.get_clauses(description)
print(clauses)
```

['First the secretary clarifies the shipment method', 'Whenever there is a requirement for large amounts', 'The secretary selects one of three logistic companies', 'Whenever small amounts are required', 'A package label is written by the secretary', 'Subsequently the goods can be packaged by the warehousemen', 'If everything is ready', 'The packaged goods are prepared for', 'Being picked up by the logistic company']

Activity recognition

```
labeled_clauses_df = d2p.activity_recognition.contains_activity_list(clauses)
print(labeled_clauses_df)
```

nr	clause	label
0	First the secretary clarifies the shipment method	True
1	Whenever there is a requirement for large amounts	False
2	The secretary selects one of three logistic companies	True
3	Whenever small amounts are required	False
4	A package label is written by the secretary	True
5	Subsequently the goods can be packaged by the warehousemen	True
6	If everything is ready	False
7	The packaged goods are prepared for	True
8	Being picked up by the logistic company	True

Activity extraction

```
extracted_activities_df = d2p.activity_extraction.get_activity_df(labeled_clauses_df)
print(extracted_activities_df)
```

nr	clause	label	activity
0	First the secretary clarifies the shipment method	True	clarify shipment method
1	Whenever there is a requirement for large amounts	False	NaN
2	The secretary selects one of three logistic companies	True	select one of three logistic companies
3	Whenever small amounts are required	False	NaN
4	A package label is written by the secretary	True	write package
5	Subsequently the goods can be packaged by the warehousemen	True	package goods
6	If everything is ready	False	NaN
7	The packaged goods are prepared for	True	prepare packaged goods
8	Being picked up by the logistic company	True	pick up

Construct semi-structured description

```
structured_description =
    d2p.structured_description.get_structured_description(description,
    extracted_activities_df)
print(structured_description)
```

first <act> clarify shipment method </act> Whenever there is a requirement for large amounts <act> select one of three logistic companies </act> Whenever small amounts are required <act> write package label </act> subsequently <act> package goods </act> If everything is ready <act> prepare packaged goods </act> <act> pick up </act>

Transform to XML format

```
xml = d2p.xml_model.structured2xml(structured_description)
print(xml)
```

<act> clarify shipment method </act> <path1> <act> select one of three logistic companies </act> </path1> <path2> <act> write package label </act> </path2> <act> package goods </act> <path1> <act> prepare packaged goods </act> </path1>

Visualization

```
image = d2p.visualization.xml2model(xml)
image
```

A.3 List of predefined conjunctions

A list of predefined conjunctions exists for individual "step" and "condition" subsentences:

- first step = ["the process starts when", "first", "initially"]
- condition = ["if", "in case", "whenever"]
- intermediate step = ["then", "next", "subsequently", "in the next step", "after that", "afterwards", "later", "thereafter", ""]
- last step = ["finally", "in the last step"]

Furthermore, a list of predefined conjunctions exists to connect two "step" subsentences:

- 2 subsequent steps = ["before", "after", "once"]

In case there are two "condition" subsentences describing an XOR split, the following conjunctions were used to connect them:

- XOR split = ["if", "in case", "whenever", "otherwise"]

A.4 Example data generation algorithm

Consider the following textual description regarding a loan approval scenario:

description = "Once a loan application has been approved by the loan provider, an acceptance pack is prepared and sent to the customer. The acceptance pack includes a repayment schedule which the customer needs to agree upon by sending the signed documents back to the loan provider. The latter then verifies the repayment agreement: if the applicant doesn't agree with the repayment schedule, the loan provider cancels the application. If the applicant agreed, he approves the application. In either case, the process completes with the loan provider notifying the applicant of the application status."

In order to create a number of semantically equivalent textual descriptions (together with their solution in XML format) from this, we first split up the description into different subsentences, each belonging to 1 out of 4 categories: 'step', 'general info', 'condition' and 'condition result'. To respect the assumptions outlined in 5.1.3, we make sure each subsentence contains at most 1 activity. In this example, the sentence "an acceptance pack is prepared and sent to the customer" contains 2 activities, so it is split up into a subsentence "an acceptance pack is prepared" and a subsentence "the acceptance pack is sent to the customer". Besides indicating the **category** for each subsentence, we also indicate a sequence number (**nr**), whether or not a conjunction is needed from the list of predefined conjunctions in A.3 (**c?**) and a reference solution for those subsentences that contain an activity in XML format (**solution**). The result is depicted in table 7 below.

Table 7: Subsentences originating from the original textual description

category	**nr**	**c?**	**subsentence**	**solution**
step	1	1	a loan application is approved by the loan provider	<act>approve loan application</act>
step	2	1	an acceptance pack is prepared	<act>prepare acceptance pack</act>
step	3	1	the acceptance pack is sent to the customer	<act>send acceptance pack</act>
general_info	4	0	the acceptance pack includes a repayment schedule which the customer needs to agree upon	
step	5	0	the customer must send the signed documents back to the loan provider	<act>send signed documents back</act>
step	6	0	the loan provider then verifies the repayment agreement	<act>verify repayment agreement</act>
condition _path1	7	1	the applicant disagreed with the repayment schedule	
condition_ result_path1	7	0	the loan provider cancels the application	<act>cancel application</act>
condition _path2	7	1	the applicant agreed	
condition_ result_path2	7	0	the loan provider approves the application	<act>approve application</act>
step	8	0	in either case, the process completes with the loan provider notifying the applicant of the application status	<act>notify application status</act>

As one can see, this textual description results in 11 different subsentences, 6 of them being "step"s, 1 "general info", 2 "condition"s (split into condition_path1 and condition_path2) and 2 "condition result"s (split into condition_result_path1 and condition_result_path2). In this case, we have a regular XOR split with one activity in each branch before merging again. Note that we use the same number for all subsentences describing an entire XOR split and merge (in this case, number 7). Next, for those obtained subsentences which can be described in a different way (by altering the active/passive tense or by using synonyms), a semantically equivalent version is added. An arbitrary number of semantically equivalent versions which are added are indicated in bold in table 8 below. Note that these have the same category, number and solution as their original form.

Table 8: Subsentences with the addition of a number of semantically equivalent versions

category	**nr**	**c?**	**subsentence**	**solution**
step	1	1	a loan application is approved by the loan provider	<act>approve loan application</act>
step	**1**	**1**	**the loan provider approves a loan application**	**<act>approve loan application </act>**
step	2	1	an acceptance pack is prepared	<act>prepare acceptance pack</act>
step	3	1	the acceptance pack is sent to the customer	<act>send acceptance pack</act>
general_info	4	0	the acceptance pack includes a repayment schedule which the customer needs to agree upon	
step	5	0	the customer must send the signed documents back to the loan provider	<act>send signed documents back</act>
step	6	0	the loan provider then verifies the repayment agreement	<act>verify repayment agreement</act>
step	**6**	**0**	**the repayment agreement is then verified by the loan provider**	**<act>verify repayment agreement</act>**
condition_ path1	7	1	the applicant disagreed with the repayment schedule	
condition_ path1	**7**	**1**	**the application did not agree with the repayment schedule**	
condition_ result_path1	7	0	the loan provider cancels the application	<act>cancel application</act>
condition_ path2	7	1	the applicant agreed	
condition_ path2	**7**	**1**	**the application accepted the repayment schedule**	
condition_ result_path2	7	0	the loan provider approves the application	<act>approve application</act>
step	8	0	in either case, the process completes with the loan provider notifying the applicant of the application status	<act>notify application status</act>
step	**8**	**1**	**the loan provider notifies the application of the application status**	**<act>notify application status </act>**

Based on this table, the data generation algorithm can now construct a large number of semantically equivalent textual process descriptions together with their solution in XML format. The Python code of two functions for this algorithm are outlined below. The first function shows how a description is generated by sequentially adding all types of sentences. The second function shows how the first sentence of a description is generated. Equivalent functions of the second function exist for the other types of sentences but are not included in the appendix.

```
def create_text(all_ordered_clauses) :

    # initialize text and solution
    text = ""
    solution = ""
    is_first_step = True

    # Loop through all ordered clauses
    while len(all_ordered_clauses) > 0 :

        # This function checks what the next block looks like: is
        # it a step, a condition, condition result or general_info?
        # Returns the type of the next possible clauses and the row with all
        # the information over the next possible clauses
        type1, df_clause = get_type_and_prepare_block(all_ordered_clauses.pop(0))

        # Depending on the type of the class a different function is called
        if type1 == "step" :

            # Check if this is the first sentence of the process description
            if is_first_step :
              # Create a first sentence and its solution with appropriate
              # conjunction
              text_temp, solution_temp = get_first_sentence(df_clause)
              is_first_step = False

            else :
              # Check if there are 2 consecutive blocks of the type "step"
              if ( len(all_ordered_clauses) > 0) and
               ↪  (is_next_step_of_type_step(all_ordered_clauses[0]) ) :
                # A sentence and solution is constructed with two clauses of
                # the type "step"
                text_temp, solution_temp =
                 ↪  get_sentence_with_multiple_steps(df_clause,
                 ↪  all_ordered_clauses.pop(0))

              else :
                # Only one block has type "step", so there is only one option # to
                 ↪  construct a new sentence for the description
                text_temp, solution_temp = get_body_sentence(df_clause)

        # Unique functions do also exist for the different types of
        # conditions and the general info type. Depending on the type a
        # sentence and its solution are returnd by the functions below.
        elif type1 == "split_path1" :
          text_temp, solution_temp = get_split_path1(df_clause)

        elif type1 == "split_path2" :
          text_temp, solution_temp = get_split_path2(df_clause)

        elif type1 == "split_path3" :
          text_temp, solution_temp = get_split_path3(df_clause)

        elif type1 == "general_info" :
          text_temp, solution_temp = get_general_info(df_clause)

        else :
           text = text + " ERROR: in choice type of sentence."

        # Add new sentence and solution to full description and complete solution
        text = text + " " + text_temp
        solution = solution + " " + solution_temp
```

```
59
60      return text.strip(), solution
```

```
1  def get_first_sentence (df_first) :
2      # Select a random row from all the possible clauses
3      random_row = df_first.sample(n=1)
4
5      # Check if conjunction is needed
6      if is_conjunction_needed(random_row) :
7        # The list begin contains the possible conjunctions for a first sentence
8        text = random.choice(begin) + " " + random_row.iat[0,2] + "."
9        solution = random_row.iat[0,3]
10     else :
11       text = random_row.iat[0,2] + "."
12       solution = random_row.iat[0,3]
```

Consider the first iteration of the algorithm (number 1). For number 1, two semantically equivalent subsentences are defined. The algorithm first checks which type the subsentences have. Depending on this type, a different method is applied. Then the algorithm randomly chooses one of the two subsentences, for example "the loan provider approves a loan application". Next, the algorithm checks whether the chosen subsentence requires a conjunction from the list of predefined conjunctions (see A.3). As this is the case here, the data generation algorithm can choose between the conjunctions ["the process starts when", "first", "initially"], since the category of the subsentence is "step" and it is the first step in the textual description. The algorithm again chooses a conjunction randomly, for example "first". These results in the first step being described as "first the loan provider approves a loan application". In fact, this is one of the 6 possible versions:

- "The process starts when a loan application is approved by the loan provider."
- "First a loan application is approved by the loan provider."
- "Initially a loan application is approved by the loan provider."
- "The process starts when the loan provider approves a loan application."
- "First the loan provider approves a loan application."
- "Initially the loan provider approves a loan application."

The same logic is repeated for each number. In case a number describes an XOR split, the algorithm also adds <path> tags to the solution. An example of an entire semantically equivalent description together with its solution (based on the table above) generated by the algorithm is given below.

generated description = "After the loan provider approves a loan application, an acceptance pack is prepared. In the next step, the acceptance pack is sent to the customer. The acceptance pack includes a repayment schedule which the customer needs to agree upon. Next, the customer must send the signed documents back to the loan provider. The repayment agreement is then verified by the loan provider. In case the applicant disagreed with the repayment schedule, the loan provider cancels the application. If the applicant accepted the repayment schedule, the loan provider approves the application. Finally, the loan provider notifies the applicant of the application status."

generated solution = <act> approve loan application </act> <act> prepare acceptance pack </act> <act> send acceptance pack </act> <act> send signed documents back </act> <act> verify repayment agreement </act> <path1> <act> cancel application </act> </path1> <path2> <act> approve application </act> </path2> <act> notify application status </act>

A.5 Dictionary of contractions

contractions = “ain’t”: “am not / are not / is not / has not / have not”, “aren’t”: “are not / am not”, “can’t”: “cannot”, “can’t’ve”: “cannot have”, “cause”: “because”, “could’ve”: “could have”, “couldn’t”: “could not”, “couldn’t’ve”: “could not have”, “didn’t”: “did not”, “doesn’t”: “does not”, “don’t”: “do not”, “hadn’t”: “had not”, “hadn’t’ve”: “had not have”, “hasn’t”: “has not”, “haven’t”: “have not”, “he’d”: “he had / he would”, “he’d’ve”: “he would have”, “he’ll”: “he shall / he will”, “he’ll’ve”: “he shall have / he will have”, “he’s”: “he has / he is”, “how’d”: “how did”, “how’d’y”: “how do you”, “how’ll”: “how will”, “how’s”: “how has / how is / how does”, “I’d”: “I had / I would”, “I’d’ve”: “I would have”, “I’ll”: “I shall / I will”, “I’ll’ve”: “I shall have / I will have”, “I’m”: “I am”, “I’ve”: “I have”, “Isn’t”: “is not”, “It’d”: “it had / it would”, “It’d’ve”: “it would have”, “It’ll”: “it shall / it will”, “It’ll’ve”: “it shall have / it will have”, “It’s”: “it has / it is”, “let’s”: “let us”, “ma’am”: “madam”, “mayn’t”: “may not”, “might’ve”: “might have”, “mightn’t”: “might not”, “mightn’t’ve”: “might not have”, “must’ve”: “must have”, “mustn’t”: “must not”, “mustn’t’ve”: “must not have”, “needn’t”: “need not”, “needn’t’ve”: “need not have”, “o’clock”: “of the clock”, “oughtn’t”: “ought not”, “oughtn’t’ve”: “ought not have”, “shan’t”: “shall not”, “sha’n’t”: “shall not”, “shan’t’ve”: “shall not have”, “she’d”: “she had / she would”, “she’d’ve”: “she would have”, “she’ll”: “she shall / she will”, “she’ll’ve”: “she shall have / she will have”, “she’s”: “she has / she is”, “should’ve”: “should have”, “shouldn’t”: “should not”, “shouldn’t’ve”: “should not have”, “so’ve”: “so have”, “so’s”: “so as / so is”, “that’d”: “that would / that had”, “that’d’ve”: “that would have”, “that’s”: “that has / that is”, “there’d”: “there had / there would”, “there’d’ve”: “there would have”, “there’s”: “there has / there is”, “they’d”: “they had / they would”, “they’d’ve”: “they would have”, “they’ll”: “they shall / they will”, “they’ll’ve”: “they shall have / they will have”, “they’re”: “they are”, “they’ve”: “they have”, “to’ve”: “to have”, “wasn’t”: “was not”, “we’d”: “we had / we would”, “we’d’ve”: “we would have”, “we’ll”: “we will”, “we’ll’ve”: “we will have”, “we’re”: “we are”, “we’ve”: “we have”, “weren’t”: “were not”, “what’ll”: “what shall / what will”, “what’ll’ve”: “what shall have / what will have”, “what’re”: “what are”, “what’s”: “what has / what is”, “what’ve”: “what have”, “when’s”: “when has / when is”, “when’ve”: “when have”, “where’d”: “where did”, “where’s”: “where has / where is”, “where’ve”: “where have”, “who’ll”: “who shall / who will”, “who’ll’ve”: “who shall have / who will have”, “who’s”: “who has / who is”, “who’ve”: “who have”, “why’s”: “why has / why is”, “why’ve”: “why have”, “will’ve”: “will have”, “won’t”: “will not”, “won’t’ve”: “will not have”, “would’ve”: “would have”, “wouldn’t”: “would not”, “wouldn’t’ve”: “would not have”, “y’all”: “you all”, “y’all’d”: “you all would”, “y’all’d’ve”: “you all would have”, “y’all’re”: “you all are”, “y’all’ve”: “you all have”, “you’d”: “you had / you would”, “you’d’ve”: “you would have”, “you’ll”: “you shall / you will”, “you’ll’ve”: “you shall have / you will have”, “you’re”: “you are”, “you’ve”: “you have”

A.6 List of special tokens for integer-encoding

All the predefined conjunctions that are outlined in A.3, are also special tokens for the integer data generation. These tokens are positioned in order to create a logical structure within an integer sequence.

Activity tags are part of a semi-structured description and are therefore also seen as a special token for the integer-encoding. In the solution, not only the activity tags occur but also the path tags. Subsequently, the “solution integer sequence” does contain the integer equivalent from the activity and path tags.

- activity tags = [“<act>”, “</act>”]
- path tags = [“<path1>”, “<path2>”, “<path3>”, “</act>”, “</path1>”, “</path2>”, “</path3>”]

A.7 Example data generation algorithm for integer-encoding

Consider the following example semi-structured description below. This semi-structured description can be integer-encoded but we pay special attention to the predefined tokens in A.6. The part of the tokenizer's dictionary that is applicable on this example is also shown under the example.

Semi-structured description:

The process starts when <act> request product </act>. If it is a new customer, <act> ask customer details </act>. Otherwise, <act> retrieve customer details </act>. Before <act> send invoice </act>, <act> send package </act> . Then <act> check payment </act>.

Tokenizer's dictionary (special tokens only):

- The process starts when → 7 8 9 10
- <act> → 1
- </act> → 27
- if → 23
- otherwise → 26
- before → 22
- then → 13

Now, the dictionary can be applied to the semi-structured description. Furthermore, we assume that we start integer-encoding the words that are no special tokens, from 100 onwards. This results in the following integer sequence (with special tokens shown in bold):

7 8 9 10 1 100 101 **27 23** 102 103 104 105 106 **1** 107 106 108 **27 26 1** 109 106 110 **27 22 1** 111 112 **27 1** 113 114 **27 13 1** 115 116 **27**

It should be clear that the bold numbers determine the structure or order of the process description. The integer data generation algorithm cleverly generates these type of structures. The not bold numbers are randomly chosen by the algorithm. Consequently, the meaning of these numbers do not make any sense when translated back to the corresponding English words. The integer data can only be used to learn how to recognize particular structures.

Below, we shortly show the code to generate random integer sequences.

```
def generate_sequence():
    # Initialize variables
    sequence =""
    solution =""

    # Create a random beginning (general info or begin activity)
    choice_begin = randint(1,2)
    if choice_begin == 1 :
      sequence = sequence + " " + generate_info()
    elif choice_begin == 2:
      activity_add, solution_add = generate_activity_first()
      sequence = sequence + " " + activity_add
      solution = solution + " " + solution_add

    # Add 4-8 other part to the dummy integer description
    length = randint(4,8)
    for _ in range(length) :
      choice = randint(1,10)

```

```
20        # Add normal activity to sequence -> 1-5
21        if choice <= 5 :
22          sequence_add, solution_add = generate_activity_normal()
23
24        # Add before activity to sequence -> 6
25        elif choice == 6:
26          sequence_add, solution_add = generate_activity_before()
27
28        # Add condition to sequence -> 7-8
29        elif choice <= 8 :
30          sequence_add, solution_add = generate_condition()
31
32        # Add general info to sequence -> 9-10
33        elif choice <= 10 :
34          sequence_add, solution_add = generate_info(), ""
35
36        else :
37          print("Error: No choice found to add to sequence.")
38
39        sequence = sequence + " " + sequence_add
40        solution = solution + " " + solution_add
41
42      return sequence.strip(), solution.strip()
```

List of Figures

List of Tables

References

Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling. 1: Parsing.* Prentice-Hall, 1972. ISBN 0139145567. URL http://www.worldcat.org/oclc/310805937.

Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *COLING 2018, 27th International Conference on Computational Linguistics*, pages 1638–1649, 2018.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL http://arxiv.org/abs/1409.0473.

Ian Beaver. Intelligently expand and create contractions in text leveraging grammar checking and word mover's distance. 2018. URL https://github.com/ian-beaver/pycontractions.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=944919.944966.

Robert Blumberg and Shaku Aktre. The problem with unstructured data. *Dm Review 13*, page 62, 2003.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *CoRR*, abs/1607.04606, 2016. URL http://arxiv.org/abs/1607.04606.

Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder. *CoRR*, abs/1803.11175, 2018. URL http://arxiv.org/abs/1803.11175.

Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750. Association for Computational Linguistics, 2014. doi: 10.3115/v1/D14-1082. URL http://aclweb.org/anthology/D14-1082.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

Gobinda G Chowdhury. Natural language processing. *Annual review of information science and technology*, 37(1):51–89, 2003.

Kevin Clark and Christopher D. Manning. Deep reinforcement learning for mention-ranking coreference models. *CoRR*, abs/1609.08667, 2016a. URL http://arxiv.org/abs/1609.08667.

Kevin Clark and Christopher D. Manning. Improving coreference resolution by learning entity-level distributed representations. *CoRR*, abs/1606.01323, 2016b. URL http://arxiv.org/abs/1606.01323.

Kevin Clark, Minh-Thang Luong, Christopher D. Manning, and Quoc V. Le. Semi-supervised sequence modeling with cross-view training. *CoRR*, abs/1809.08370, 2018. URL http://arxiv.org/abs/1809.08370.

Deborah Coughlin. Correlating automated and human assessments of machine translation quality. In *In Proceedings of MT Summit IX*, pages 63–70, 2003.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer Publishing Company, Incorporated, 2013. ISBN 3642331424, 9783642331428.

Elena Viorica Epure, Patricia Martín-Rodilla, Charlotte Hug, Rebecca Deneckère, and Camille Salinesi. Automatic process model discovery from textual methodologies: An archaeology case study. *Ninth International Conference on Research Challenges in Information Science, May 2015, Athens, Greece.*, 2015.

Solomon Feferman. Typical ambiguity: trying to have your cake and eat it too. *One hundred years of Russell's paradox*, pages 135–151, 2004.

Fabian Friedrich, Jan Mendling, and Frank Puhlmann. Process model generation from natural language text. In *International Conference on Advanced Information Systems Engineering*, pages 482–496. Springer, 2011.

Yoav Goldberg. A primer on neural network models for natural language processing. *CoRR*, abs/1510.00726, 2015. URL http://arxiv.org/abs/1510.00726.

Yoav Goldberg and Graeme Hirst. *Neural Network Methods in Natural Language Processing*. Morgan & Claypool Publishers, 2017. ISBN 1627052984, 9781627052986.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Jan Goyvaerts. *Regular Expressions: The Complete Tutorial*. Lulu Press, 2006.

Luheng He, Kenton Lee, Mike Lewis, and Luke Zettlemoyer. Deep semantic role labeling: What works and what's next. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 473–483. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1044. URL http://aclweb.org/anthology/P17-1044.

S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Matthew Honnibal and Mark Johnson. An improved non-monotonic transition system for dependency parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1373–1378, Lisbon, Portugal, September 2015. Association for Computational Linguistics. URL https://aclweb.org/anthology/D/D15/D15-1162.

Dan Jurafsky and James H Martin. *Speech and language processing*, volume 3. Pearson London, 2018.

Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188, 2014. URL http://arxiv.org/abs/1404.2188.

Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014. URL http://arxiv.org/abs/1408.5882.

D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, December 2014.

Nikita Kitaev and Dan Klein. Constituency parsing with a self-attentive encoder. *CoRR*, abs/1805.01052, 2018. URL http://arxiv.org/abs/1805.01052.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. Opennmt: Open-source toolkit for neural machine translation. In *Proc. ACL*, 2017. doi: 10.18653/v1/P17-4012. URL https://doi.org/10.18653/v1/P17-4012.

Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. From word embeddings to document distances. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 957–966. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045221.

Kenton Lee, Luheng He, and Luke Zettlemoyer. Higher-order coreference resolution with coarse-to-fine inference. *CoRR*, abs/1804.05392, 2018. URL http://arxiv.org/abs/1804.05392.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015. URL http://arxiv.org/abs/1508.04025.

Minh-Thang Luong, Eugene Brevdo, and Rui Zhao. Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*, 2017.

Xuezhe Ma and Eduard H. Hovy. End-to-end sequence labeling via bi-directional lstm-cnns-crf. *CoRR*, abs/1603.01354, 2016. URL http://arxiv.org/abs/1603.01354.

Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. *CoRR*, abs/1708.00107, 2017. URL http://arxiv.org/abs/1708.00107.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013. URL http://arxiv.org/abs/1301.3781.

Christopher Olah. Understanding lstm networks, 2015. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Hiroki Ouchi, Hiroyuki Shindo, and Yuji Matsumoto. A span selection model for semantic role labeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1630–1642. Association for Computational Linguistics, 2018. URL http://aclweb.org/anthology/D18-1191.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

Maximilian Riefer, Simon Felix Ternis, and Tom Thaler. Mining process models from natural language text: A state-of-the-art analysis. *Multikonferenz Wirtschaftsinformatik (MKWI-16), March*, pages 9–11, 2016.

Brandon Rohrer. How deep neural networks work, 2017. https://www.youtube.com/watch?v=ILsA4nyG7I0&t=1148s, Last accessed on 2018-11-07.

A. Sanfeliu and K. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):353–362, May 1983. ISSN 0018-9472. doi: 10.1109/TSMC.1983.6313167.

Konstantinos Sintoris and Kostas Vergidis. Extracting business process models using natural language processing (nlp) techniques. *IEEE 19th Conference on Business Informatics*, 2017.

Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014. URL http://arxiv.org/abs/1409.3215.

Joseph Sànchez, Han van der Aa, Josep Carmona, and Lluís Padróa. Aligning textual and model-based process descriptions. *ScienceDirect*, 2018. URL https://www.sciencedirect.com/science/article/pii/S0169023X18301496?via%3Dihub.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. 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)*, pages 1556–1566. Association for Computational Linguistics, 2015. doi: 10.3115/v1/P15-1150. URL http://aclweb.org/anthology/P15-1150.

Lucinéia Heloisa Thom, Vinicius Stein Dani, Thanner Soares Silva, and Marcelo Fantinato. Natural language processing in business process identification and modeling: A systematic literature review. *Conference: XIV Simpósio Brasileiro de Sistemas de Informação (SBSI)*, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Lukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018. URL http://arxiv.org/abs/1803.07416.

Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2773–2781. Curran Associates, Inc., 2015. URL http://papers.nips.cc/paper/5635-grammar-as-a-foreign-language.pdf.

Xin Wang, Yuanchao Liu, Chengjie SUN, Baoxun Wang, and Xiaolong Wang. Predicting polarities of tweets by composing word embeddings with long short-term memory. 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)*, pages 1343–1353. Association for Computational Linguistics, 2015. doi: 10.3115/v1/P15-1130. URL http://aclweb.org/anthology/P15-1130.

Thomas Wolf. The current best of universal word embeddings and sentence embeddings. 2018. https://medium.com/huggingface/universal-word-sentence-embeddings-ce48ddc8fc3a.

Chuyu Xiong. What really is deep learning doing? *CoRR*, abs/1711.03577, 2017. URL http://arxiv.org/abs/1711.03577.

Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *CoRR*, abs/1708.02709, 2017. URL http://arxiv.org/abs/1708.02709.