Speech To Code
- The Game Changer -

TDDE19 - Report

Henrik Andersson
henan076@student.liu.se
Mathias Berggren
matbe320@student.liu.se
Viktor Agbrink
vikag322@student.liu.se

Anton Gefvert
antge210@student.liu.se
Daniel Sonesson
danso829@student.liu.se
Antymos Dag
antda166@student.liu.se

December 2019

1 Introduction

Common injuries among working programmers are related to the repetitive nature of typing on a keyboard, these injuries include: repetitive strain injuries, and carpal tunnel. One way of solving this issue is to use a different input method, such as voice. Previous work on using voice as the input method for coding required the programmer to memorize keywords which mapped to specific actions or modes. This essentially forces the programmer the learn another language to use while programming.

The goal of this project is to investigate and prototype an input system which allows a programmer to program using natural language. This project is part of the course *TDDE19 - Advanced Project Course in AI and Machine Learning*, and the problem will therefore be modeled as a machine learning problem. Natural language processing problems have been modeled as machine learning problems to much success using neural network methods such as: recurrent networks, convolutional networks, and transformer networks.

Going from natural language to code presents various numbers of challenges. Such as the intent analysis of the input sentence, finding variables and numbers and also generating the actual code. The report presents our take on this issue and our solutions to these problems.

1.1 Aim

The aim of the project is for a programmer to be able conduct his work for one whole hour without having to touch either a mouse or a keyboard but instead only use his voice. Thus, this also includes being able to navigate files and reading documentation, in addition to programming.

1.2 Delimitations and Assumptions

In order to keep the scope of the project to a reasonable size, the *Speech-to-text* (STT) part of the project has been left out. As so, perfect text input from an STT is assumed throughout the project.

The target programming language for the project was decided to be Python, as all the authors had experience with this and as it was regarded as a fairly syntactically simple language.

Atom was chosen as the target IDE for this project. This is because Atom has an extensive API that suited our needs.

2 Method

The challenge of going from natural language to code was regarded as a translation problem. Meaning that we wanted to take a sentence in natural language and translate it to code. Figure 1 shows an overview of the different stages that was included to solve this translation problem.

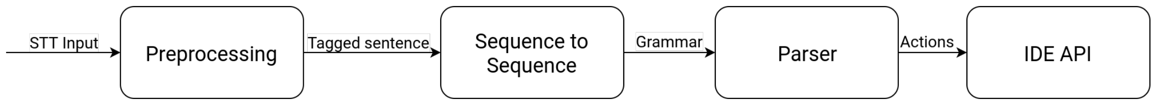


Figure 1: Overview of the project stages

It starts with a preprocessing stage where each input-sentence is tagged in order to find the variables, numbers etc. in the sentence. The tagged sentence is then passed to a Sequence-to-Sequence stage, where it is translated to a sequence of our self-defined grammar. This grammar is then parsed to actions that is sent to the selected IDE and its API. The following sections gives a more detailed view into each step.

2.1 Pre-processing

A Name Entity Recognition (NER) model is used to pre-process our input. The task of the NER model is to classify named entity mentions in text into predefined tags. These are later used in our Sequence-To-Sequence model, which translate fluid text into a predefined grammar. The tags that we use in our NER model is the following:

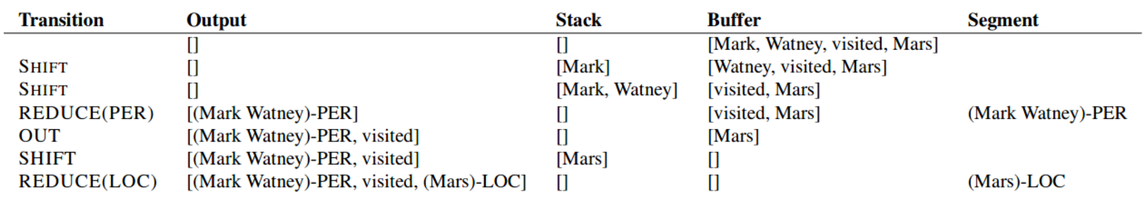
- Variable: VAR
- Number: NUM
- None: NON
- Boolean: BOOL
- String: STR
- Logarithmic operations: LOG
- Other: O

These tags also follow a IOB-structure when needed. The IOB-structure is a way to represent separate words that is of the same type of label. An example of this is represented in Table 1 below:

Sentence	**Tags**
add v max with zero point four as default	O B-VAR I-VAR O B-NUM I-NUM I-NUM O O
if variable name is equal to Daniel	O O VAR O O O STR

Table 1: Example sentences and its corresponding tags

In Table 1 it is noticable that the first sentence use the mentioned IOB-structure. That is because the string *v max* shall output one variable *v_max* and the string *zero point four* shall output the number *0.4*. Worth to mention is that the second sentence does no follow this structure, this is simply because it is not needed.



Transition	**Output**	**Stack**	**Buffer**	**Segment**
	[]	[]	[Mark, Watney, visited, Mars]	
SHIFT	[]	[Mark]	[Watney, visited, Mars]	
SHIFT	[]	[Mark, Watney]	[visited, Mars]	
REDUCE(PER)	[(Mark Watney)-PER]	[]	[visited, Mars]	(Mark Watney)-PER
OUT	[(Mark Watney)-PER, visited]	[]	[Mars]	
SHIFT	[(Mark Watney)-PER, visited]	[Mars]	[]	
REDUCE(LOC)	[(Mark Watney)-PER, visited, (Mars)-LOC]	[]	[]	(Mars)-LOC

Figure 2: Transition based state machine used by the NER model [1].

The NER-model used in this project is based on the Python NLP-framework Spacy. This model is inspired by Lample et al. (2016) transition based state machine which is represented in Figure 2 above [1]. The process start with an empty stack, with all words in a buffer and no entities, then an action is defined that change the state. To perform the correct action, the model make predictions given its current state. These predictions is made possible due to Spacy's pipeline: *Embed, Encode, Attend, Predict.* Which is represented in Figure 3 below [2].

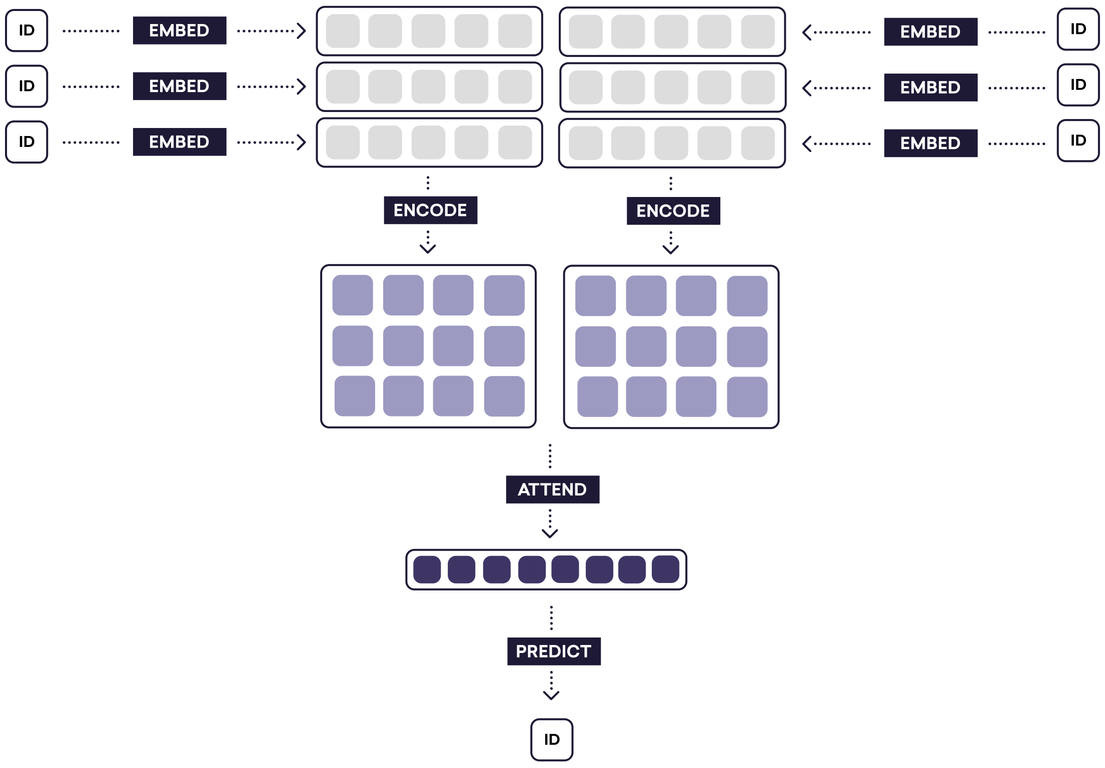


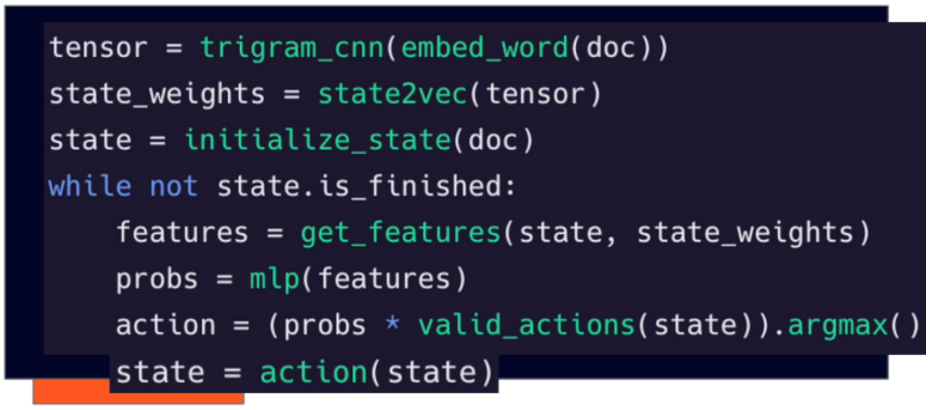
Figure 3: The Spacy's NER model pipeline [2].

In the *Embed* part of the pipeline words get represented as vectors and each of the words got four features:

- NORM: The word in normalized form.
- PREFIX: The prefix of the word
- SUFFIX: The suffix of the word
- SHAPE: The shape of the word, this can for instance be d for digits w for lower case char and W for upper case char.

To represent these features as a vector, each feature gets concatenated and feed through a multi-layer perceptron containing one hidden layer and a maxout unit. After the *Embed* part of the pipeline, the *Encode* stage transform these context-independent vectors into a context-sensitive matrix. This is made possible with a Convolutional Neural Network, containing a context-window with one neighboring word on each side of the current word. After the convolution step, the vector is now of dimensions tree times as large as the original vector, due to the context extracted from its neighboring words. This vector is then feed through a residual multilayer perceptron with a maxout layer. Returning a vector of the same original dimensionality, but with the context of the neighboring words taken into consideration. This step is then performed four times, which gives a vector affected by its four closest neighbors. After the *Encode* part of the pipeline *Attend* is reach. In this step the matrix gets reduced from the representation obtained in the previous step into a single vector, which is called a weighted summary vector and is later used for prediction. This is made possible by performing matrix-operations with the context-sensitive matrix and a context vector representing the current state in the state machine. When this is done, the weighted summary vector is feed through a multilayer perceptron. This later output the class tags as real valued numbers. These values is later parsed to find out which tag-value to assign for the current word [2]. That is how the

Spacy NER-model works, a psuedo-code representation of the process is displayed in Figure 4 below [2].



```
tensor = trigram_cnn(embed_word(doc))
state_weights = state2vec(tensor)
state = initialize_state(doc)
while not state.is_finished:
    features = get_features(state, state_weights)
    probs = mlp(features)
    action = (probs * valid_actions(state)).argmax()
    state = action(state)
```

Figure 4: Psudo code for the Spacy's NER model [2].

The last step in the pre-processor was to translate string representations of numbers into numerical representations. This was made possible due to the use of the Python library Word2Num (w2n) which transform string represented words into numerical ones. This process is represented in Table 2 below.

Sentence before (w2n)	**Tags before (w2n)**
add v max with zero point four as default	O B-VAR I-VAR O B-NUM I-NUM I-NUM O O
Sentence after (w2n)	**Tags after (w2n)**
add v max with 0.4 as default	O B-VAR I-VAR O NUM O O

Table 2: Example sentences and tags before/after w2n.

2.2 Grammar

Since the problem was going to be modeled as a translation problem, a target language had to be defined to which English was going to be translated. This language needed to capture a subset of the most common actions made while coding, to allow the programmer to not have to switch into a non-natural language input frequently. The reasoning for capturing only a subset of actions was that other actions could be captured more easily through context words.

To define the target language a strict grammar was defined, which can be seen in Figure 5. The grammar is defined to allow for some common editor actions such as: moving the cursor, and removing/adding lines, words, or indentations. It also allows for the most common coding inputs for python, such as: definitions, adding parameters/keys/values, control statements, and expressions.

For this project the grammar needed to be restricted such that the prediction space did not grow too large. The prediction space needed to be limited since all training data was going to be generated by the project members.

2.3 Sequence-To-Sequence

To translate natural language into the defined grammar described in section 2.2, a Sequence-To-Sequence(s2s) model, which takes natural language as input and outputs a sentence in the defined grammar, was implemented. A level description of this model is as follows:

```
dir := prev | next
movement := (line | char | word | indent) [NUM] [dir]
action := (delete | insert | copy | paste | move) [...movement]

tag := O | VAR | NUM | NON | BOOL | STR | LOG
ref := tag | (reference | call) VAR | (index | property) ref ref

conj := or | and | is | in | < | <= | >= | > | == | + | - | * | / | ** | %

expr := ref | conj expr expr | not expr

def := define (func | class | list | dict | variable) ref [expr]

add := add (parameter | key | value) expr [default expr]

control := (if | elif | else | while | for | return | yield) [expr]
```

Figure 5: The grammar defining the target language. Notation: ":=" definition, "a | b" a or b, "[a]" optional a, "...a" one or more a.

1. Get natural language input
2. Pre-process input with different tags, replace the words with the tags, and save words replaced by tags
3. Send the pre-processed sentence into a LSTM, word by word (as described in Section 2.1).
4. Discard the output of the first LSTM and send the hidden layer parameters to a second LSTM
5. Using the second LSTM, give a *start-of-sentence* word as first input to predict the first word in our grammar (as described in Section 2.2)
6. Feed the predicted word back into the second LSTM and predict the next word, removing any predictions that do not follow the grammar
7. Repeat step 6 until a *end-of-sentence* (eos) word is predicted.
8. Concatenate all words outputted by the second LSTM (except eos)
9. Replace any outputted tags with the saved words from step 2

2.3.1 Encoder-Decoder

The model used is a Encoder-Decoder model as illustrated in Figure 6. The Encoder-Decoder model is implemented using two LSTMs with the following properties:

- **Encoder** - The *Encoder* takes natural language input and discards the input, the important part is the hidden values that are passed on to the *Decoder*.
- **Decoder** - The *Decoder* initializes its' hidden values to the values sent by the *Encoder*, it then takes a *start-of-sentence* tag as first input and outputs a predicted value based on the input and hidden values, this output is then fed back as input and this continues until a *end-of-sentence* tag is predicted.

The initial implementation was done following the guide described in [3], tweaked to work on our data.

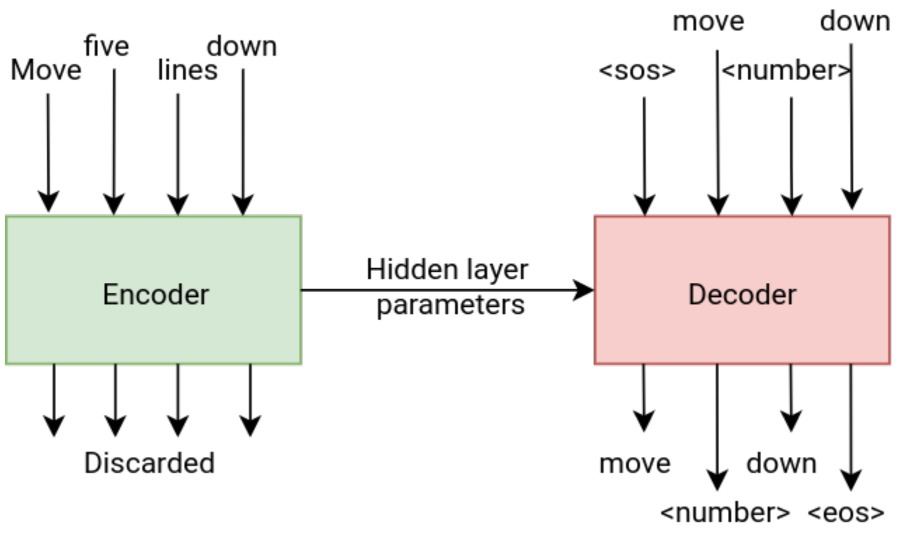


Figure 6: Encoder-Decoder model

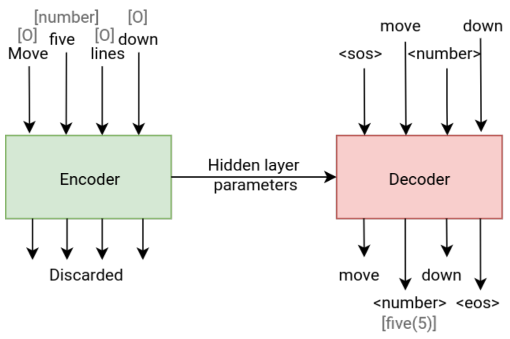
2.3.2 Modifications

During implementation two major problems with the above mentioned model was identified:

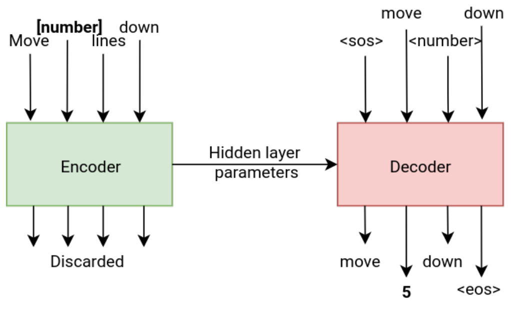
1. *Variable names, numbers, et cetera.* Since programming uses a lot of numbers and custom names for entities, the input vocabulary cannot feasibly contain all of these words. There is also the problem of needing these words in the output, meaning the output vocabulary also would have to contain these words.
2. *Strict grammar.* Even if the output vocabulary only contains words defined in our grammar, the *Decoder* could predict any combination of these words, but only a small subset of these combinations are valid according to the grammar.

To solve these problems the following solutions was implemented

Tag-pre-processing To avoid the problem of having unnecessary amount of words in both our input and output vocabulary, a pre-processing step to tag input words, as described in Section 2.1, was implemented. In Figure 7, an example of this process is illustrated. As seen in Figure 7a, tags are predicted for each word in the input sentence, the values for all words predicted by any tag other than [O] is saved. The model can predict words that correspond to the possible pre-processing-tags (in this example *<number>*). After the words in the input are tagged, any non-[O] tagged word is replaced by its tag, as seen in Figure 7b, prediction is performed and any tag present in both input and output is replaced by the original value of that word.



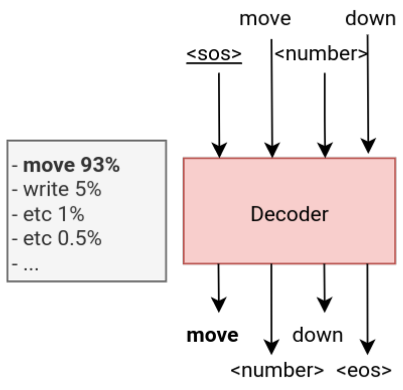
(a) Tagged words



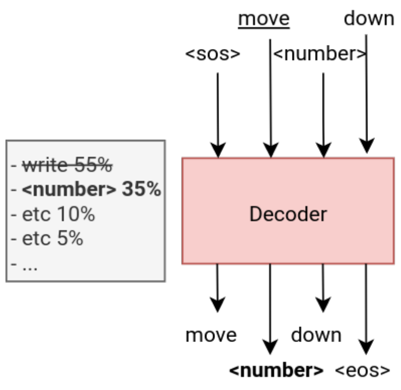
(b) Replaced words

Figure 7: Tag pre-processing used in Sequence-To-Sequence model

Output-pruning To ensure all output of the s2s model is of correct grammar, an output-pruning module was implemented. This module was added to the *Decoder*-step of the s2s model, where given a input word, it outputs a list of possible output words as defined in the grammar, any non-valid word *cannot* be predicted. This process is illustrated in Figure 8, where the underscored word is the input and the bold is the predicted word. Figure 7b shows an invalid prediction (write) with the highest prediction probability, but since it is invalid, the second highest prediction (number in our case) is used instead.



(a) Word prediction



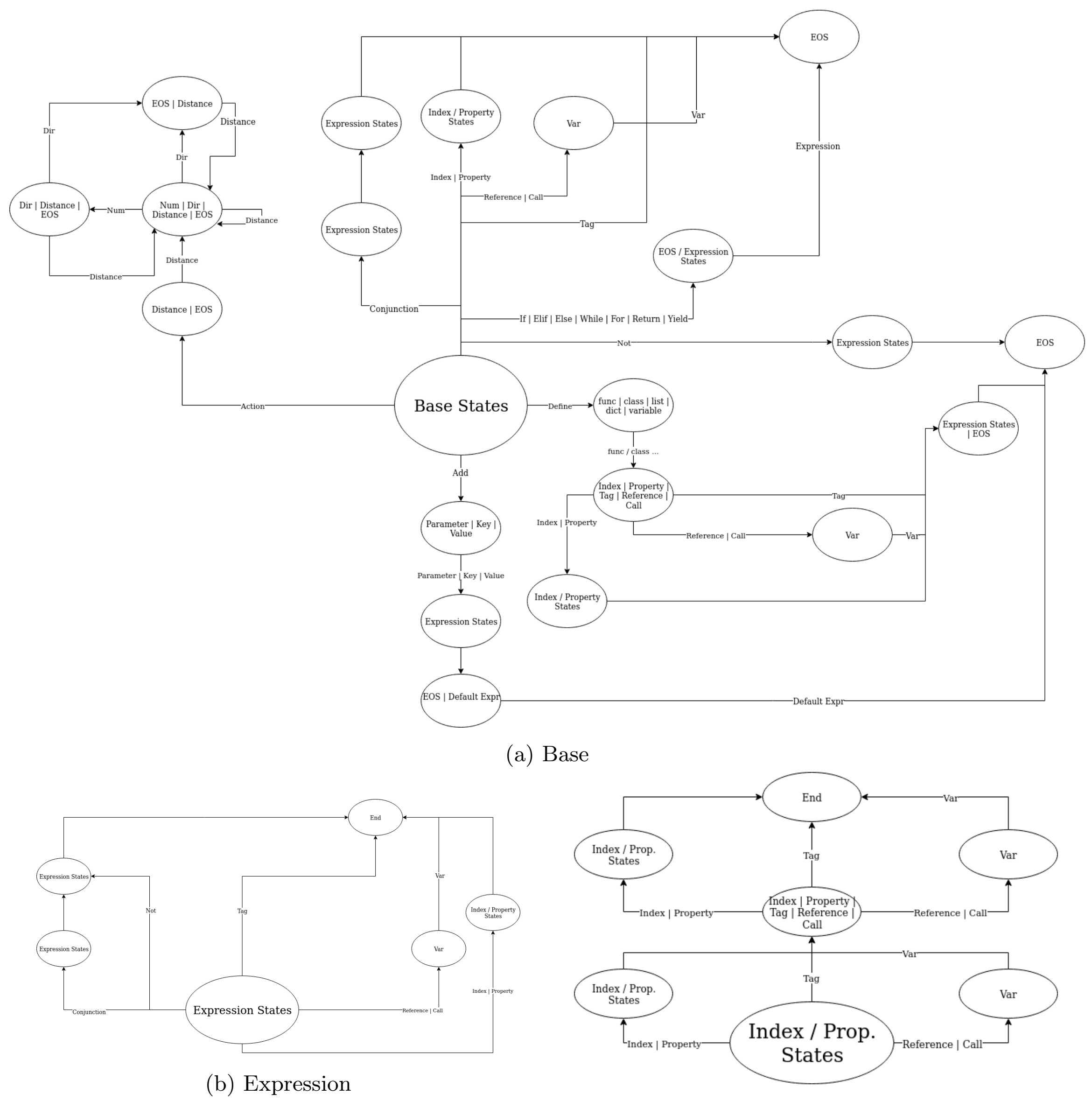
(b) Pruned word prediction

Figure 8: Output-pruning in the Decoder

2.3.3 Output-pruning Module

As mentioned earlier an output-pruning module was used to only let the LSTM output valid grammar. The problem is similar to a parsing problem in the sense that it can be modeled with a finite automata (finite state machine). The state machine is then used to keep track of which state that is currently active, and what words that can be used in the current state to proceed into a new state.

Since the defined grammar allows infinite sentences, there was a need to create a state machine that allows recursion into itself. This is used to keep track of; the number of expressions that has currently been processed in a conjunction, and how many times a reference has been selected for index/properties. This was solved by creating several state machines. By looking at the defined grammar in Figure 5 the conclusion was that this could happen in two places, expressions nestled by conjunctions and index/propertys in the reference tree. Therefor both of the mentioned sub-grammar-trees got a separate state machine. In Figure 9 the final structure can be viewed.



(a) Base

(b) Expression

(c) Index / Property

Figure 9: Output-pruning module modeled as several state machines

2.3.4 Baseline

To have some comparable metric for how well the s2s solution works, a baseline was implemented. Because of the varying length nature of a LSTM solution, a combined solution of a naive-bayes word predictor and a most common sentence baseline was created. The naive-bayes part makes use of the nature of the defined grammar, since there is only a small number of words that a output sentence is allowed to start with. The naive-bayes model takes a input as a bag of word representation and and output one of these possible starting words. After a starting word is predicted, the model takes the most common occurring sentence in the training data that starts with the predicted word, and "predicts" that sentence.

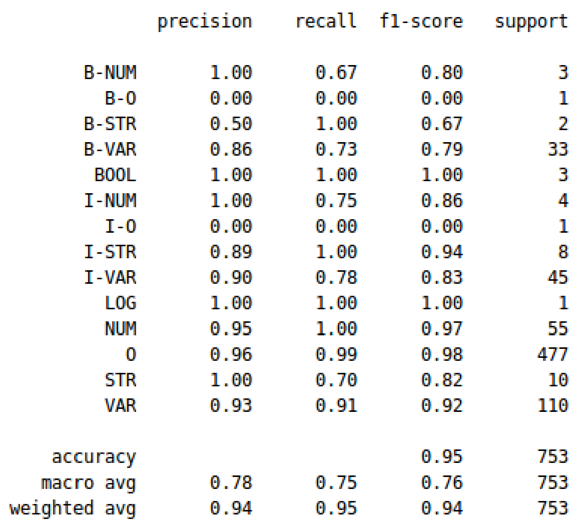
3 Results

3.1 Pre-processing

The hyperparameters used by the NER-model was evaluated using grid search, on our 600 data points, divided into a (80%/20%) train/test-split. The hyperparameters that was taken into consideration was the following:

- Number of epochs: [10,50,100,500]
- Batch size: [4,8,16,32,64]
- Drop out values: [0.1,0.2,0.3,0.5,0.8]

The resulting optimal parameters obtained from the grid search was: number of epochs = 100, batch size = 32 and the drop-out value = 0.3. These values gave a surprisingly accurate model with an sentence accuracy of 82% and a word accuracy of 95%. To put these numbers into perspective we compared it to a baseline where the model predicted all of the tags as O (other). The baseline gave a sentence accuracy of 9.9% and a word accuracy of 63%. The performance of each tag was the following values presented in Figure 10.



```
              precision    recall  f1-score   support

       B-NUM       1.00      0.67      0.80         3
         B-O       0.00      0.00      0.00         1
       B-STR       0.50      1.00      0.67         2
       B-VAR       0.86      0.73      0.79        33
        BOOL       1.00      1.00      1.00         3
       I-NUM       1.00      0.75      0.86         4
         I-O       0.00      0.00      0.00         1
       I-STR       0.89      1.00      0.94         8
       I-VAR       0.90      0.78      0.83        45
         LOG       1.00      1.00      1.00         1
         NUM       0.95      1.00      0.97        55
           O       0.96      0.99      0.98       477
         STR       1.00      0.70      0.82        10
         VAR       0.93      0.91      0.92       110

    accuracy                           0.95       753
   macro avg       0.78      0.75      0.76       753
weighted avg       0.94      0.95      0.94       753
```

Figure 10: Results for each tag predicted by Spacy's NER model

3.2 Sequence-To-Sequence

In Table 3 the results of the baseline and Encoder-Decoder model with regard to sentence accuracy (amount of sentence completely correctly guessed) and word accuracy (amount of words predicted in the correct positions), is presented.

The results are based on training performed on 774 data-points. The data for the Encoder-Decoder model is split in train-val-test data in a 60-20-20 split. The data for the baseline is split into 80-20 train-test split.

To get an better understanding of how well the baseline works, the metrics for the Naive-Bayes word prediction model is presented in Table 4.

	Baseline	Encoder-Decoder
Sentence accuracy	23.2%	53.2%
Word accuracy	60.6%	73.9%

Table 3: Sequence-To-Sequence results

	Accuracy	Precision	Recall	F1-Score
Naive-Base word prediction	94%	95%	93%	0.94

Table 4: Naive-Bayes word prediction results

4 Discussion

4.1 Pre-processing

The performance of the model was very good in this case, but it is also worth to mention that this performance is rather limited to problem specific sentences. This is largely due to the low amount of training data used to train the model. Thus, it is suspected that the performance would be worse on out-of-distribution samples. Although the performance was good, one might argue that the performance can become even better. For example, if we would remove all of the original tags and represent them as B-tags, then we would narrow down our search-space from 19 different possible outcomes, down to 13 possible outcomes which is almost a reduction by 30%.

4.2 Sequence-To-Sequence

Looking at the results for the *Encoder-Decoder* model with the small amount data given, the results are very promising. It predicts slightly more than half of all sentences correctly and has almost correct on 75% of all words. This shows that the model is relatively close to predicting correctly on most sentences. There is also a problem with both these metrics, for example if we predict one word to much or to little, any following words will count as a mis-predicted words. An example of this could be the following:

- Correct: *for in VAR reference VAR*
- Predicted: *for in reference VAR reference VAR*

In this example we would have a word-accuracy of 40%, even though it is very close to being correct.

Even though the dataset is very small and biased (due to being created by only six different people), there is reason to believe this model could actually prove useful. This was indicated by the observation of some degree of generalization in e.g. the ability to distinguish the intended direction of navigation movement given some input sentence. The main contribution necessary to create a practically useful system would be to create a much larger dataset featuring a varied syntactic style (i.e. choice of words) in the input sentences. This would alleviate the bias issues and allow for a greater degree of generalization. Since labeling data is relatively expensive, it might prove fruitful to extend the system with a component for active learning. Active learning would allow the system to interactively query for labels to unlabeled samples during training, allowing for active selection of which data points are the most valuable.

4.3 Grammar

The structure of the grammar has an impact on the complexity of the search-space in both pre-processing and sequence-to-sequence translation. It dictates which tokens are allowed in the output (via output-pruning). It also dictates which actions are possible in the development environment and the manner in which some aggregate actions are performed. Therefore, it is meaningful investigate how the grammar might affect the performance of the system. In order to assess how a grammar impacts the structure, one would have to compare the performance of the system several using various grammars. Although, this is was out of the scope of this project. One possible direction for future work would be to construct a new grammar and compare it to the existing one in terms of performance.

Another potential extension to this project would be to investigate how it would affect the grammar if one was to attempt to support several possible target programming languages. The degree to which the grammar is tailored to a specific programming language might become more clear by doing this. For example, allowing the use of several programming languages which support different programming paradigms could potentially prove to be very challenging.

5 Conclusion

The results of this project are both positive and negative. The individual results from each stage shows great promise when tested using the training/test data that was generated. Especially the preprocessing stage which had 82% accuracy on whole sentences. However, combining all stages into a complete system, where a user can give their own input, it quickly reveals some flaws. The system is able to correctly translate and parse easy inputs, such as cursor movement and simple variable declarations. But when traversing beyond that, into more complex inputs, the translations starts to dwindle. Making the system more or less unusable in the current stage. Unfortunately, due to time constraints, there is no numerical data depicting the results for the whole system.

The cause for this is the amount of available data the project had. Since no data was available beforehand it had to be generated. Although some data was generated, roughly 800 data points, it is not nearly enough to cover all instances of possible inputs. Thus, it greatly limits the amount of generalization that our solutions can do. As the data is only generated by six people it is also over fitted to the way those six people express themselves.

The project shows great promise when looking at the individual parts of the system, which tells us that with access to more data the aim of the project should be reasonable.

References

[1] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, “Neural architectures for named entity recognition”, *arXiv preprint arXiv:1603.01360*, 2016.

[2] M. Honnibal, “Spacy’s entity recognition model”, 2017. [Online]. Available: `https://spacy.io/models`.

[3] *A ten-minute introduction to sequence-to-sequence learning in keras*, `https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html`, Accessed: 2019-12-19.