Automatic role labeling

Emotion analysis – assignment 4

Carlotta Quensel
Felix Bühler
Maximilian Wegge

February 7, 2021

1. Introduction

As the task of sequence labelling is more advanced than whole-text emotion classification, we expect there to be more difficulties in automatic labelling as well as bigger differences between naïve and complex algorithms.

Thus, we decided to do automatic role labeling to confirm these expectations. The sequences for emotion experiencer, target or stimulus are mostly semantically determined and do not directly map onto syntactic structures, but the difference in syntactic structure between different training data might still hold an effect on the results. In this assignment, we look to answer if and how a naive and complex sequence labelling approach differ, as well as compare the influence of different training data on the results. With these objectives as our research questions, we now are able to decide on our method and data.

2. Method

Both methods are trained on the same data, so that we can compare the intersection between data and algorithm influences. Therefore, our first step in implementing a role labelling algorithm is to decide on the different data.

2.1. Training data

To keep the task mamageable, we only label one role, which is target. The semantic role of emotion target might in a naïve understanding correspond to the syntactic role of object (e.g. *I am angry at* ***you***.). This is of course not correct in many instances but still opens an interesting distinction between corpora. As the Reman corpus consists

of literary texts, GoodNewsEveryone of news headlines and Electoral tweets of tweets, these three corpora have very different syntactic styles, the news being abbreviated and the literature containing more complex syntactic contructs. While the distribution of our selected role is not ideal between the corpora (Reman only contains around 700 target instances), going forward, these are the three corpora used for training and evaluating both algorithms.

2.2. Naïve approach – Hidden Markov model

Sequence labelling in general and the role labelling of emotion target specifically is dependent on the word order, sentence semantic and therefore also syntactical information. While both approaches only consider the tokens as labelling information, while the complex method learns to hopefully detect underlying structures in the token order, the simple method only considers the order of labels. As a basic sequence labelling method, we chose to train a Hidden Markov model and then determine the best label sequence using a viterbi algorithm (the code can be found at the end of this documentation). The model is trained with the token-label pair frequencies relative to the token frequency as emission probabilities and the label bigram frequencies relative to the second token as transition probabilities. An estimated best label sequence therefore combines the most probable label sequence with the highest possibilities of labels for each individual token. The viterbi algorithm is then used to compute this most probable label sequence for a given token sequence.

While Hidden Markov models have many applications in natural language processing, as a sequence labelling algorithm, it is most frequently used for POS-tagging. In contrast to POS-tagging, a HMM in our case has a deficit, as the tokens are not connected to the tags as strongly. While a token can change its syntactical category according to its place in the sentence, there are generally only a few possible labels it can take on. For emotion role labelling on the other hand, a sequence like *my mother* can be the emotion target as well as the experiencer or even the stimulus. The classification mostly depends on semantic information which might be transported through syntactic structure, which our approach does not take into account, as the tokens themselves are only counted as unigrams. Thus, we use a more complex learning approach to combat the problem of missing context.

2.3. Complex approach – Transformer

For the complex approach, we chose a deep learning algorithm. The used transformer takes the token sequence as an input matrix and therefore can use the whole sequence as context for each individual token.

3. Evaluation

To evaluate our methods, we count intersections between the gold and the predicted target sequence as correct labels (True Positives). When a gold label sequence spans more than one predicted sequence or the other way around, we only count one of these multiples and ignore the rest.

		Gold	
		In	Out
Pred	In		
	Out		

3.1 Confusion matrix of the target sequences predicted by the Hidden Markov model trained on the Reman data

A. Code

A.1. HMM and Viterbi

```
import pandas


class HiddenMarkovModel:
    transitions = dict()
    emissions = dict()
    prior = dict()

    def __init__(self):
        self.transitions = {'O': {'O': 0, 'B': 0, 'I': 0},
                            'B': {'O': 0, 'B': 0, 'I': 0},
                            'I': {'O': 0, 'B': 0, 'I': 0}}
        self.prior = {'O': 0, 'B': 0, 'I': 0}

    def train_probabilities(self, train_set: pandas.DataFrame):
        trans = {'O': {'O': 0, 'B': 0, 'I': 0},
                 'B': {'O': 0, 'B': 0, 'I': 0},
                 'I': {'O': 0, 'B': 0, 'I': 0}}
        emi = dict()
        prior = {'O': 0, 'B': 0, 'I': 0}
        for index, row in train_set.iterrows():
            # The current sequence consists of (word, tag) pairs
            seq = list(zip(row['tokens'], row['target']))
```

```
            for i in range(len(seq)):
                tok = seq[i][0]
                tag = seq[i][1]
                # First tag: Prior count
                if i == 0:
                    prior[tag] += 1
                # Every other tag: Transition count from the previous tag
                else:
                    trans[tag][seq[i - 1][1]] += 1
                # Emission count from token-tag-pairs
                if tok in emi:
                    emi[tok][tag] += 1
                else:
                    emi[tok] = {'O': 0, 'B': 0, 'I': 0}
                    emi[tok][tag] += 1
        # Converting the absolute emission/transition/prior frequencies in
        for tag in {'O', 'B', 'I'}:
            freq = sum(trans[tag].values())
            for prev_tag in trans[tag]:
                trans[tag][prev_tag] /= freq
            prior[tag] /= sum(prior.values())

        for tok in emi:
            freq = sum(emi[tok].values())
            for tag in {'O', 'B', 'I'}:
                emi[tok][tag] /= freq
        # Unknown words get a random tag
        emi['$OOV'] = {tag: sum(trans[tag].values()) /
                        sum([sum(trans[tag2].values())
                             for tag2 in {'O', 'B', 'I'}])
                        for tag in {'O', 'B', 'I'}}
        self.transitions = trans
        self.emissions = emi
        self.prior = prior


def viterbi(sequence: list[str], model: HiddenMarkovModel) -> list[str]:
    viterbi_table = [[0 for _ in range(3)] for _ in range(len(sequence))]
    max_labels = list()
    # Go through the sequence token by token
    for i_tok, token in enumerate(sequence):
        # For each possible Tag, multiply
        for i_tag, tag in enumerate(['O', 'B', 'I']):
            # the emission probability of the tag and current token
```

```
            if token in model.emissions:
                viterbi_table[i_tok][i_tag] = model.emissions[token][tag]
            else:
                viterbi_table[i_tok][i_tag] = model.emissions['$OOV'][tag]
            # with the prior probability of the tag for the first token
            if i_tok == 0:
                viterbi_table[i_tok][i_tag] *= model.prior[tag]
            # or with the transition probability for best tag of the last
            else:
                viterbi_table[i_tok][i_tag] *= max_v * model.transitions[t
        max_v = max(viterbi_table[i_tok])
        max_t = ['O', 'B', 'I'][viterbi_table[i_tok].index(max_v)]
    indices = {0: 'O', 1: 'B', 2: 'I'}
    # Read the best tags from the table and return the sequence
    for i in range(len(sequence)):
        max_labels.append(indices[viterbi_table[i].index(max(viterbi_table
    return max_labels
```