BACHELOR'S THESIS

Degree in Mathematics

Title: Introduction to Natural Language Understanding and Chatbots.

Author: Víctor Cristino Marcos

Advisor: Jordi Saludes

Department: Matemàtiques (749)

Academic year: 2017 - 2018

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Introduction to Natural Language Understanding and Chatbots

Víctor Cristino Marcos

BACHELOR'S THESIS

supervised by
Jordi Saludes Closa

Universitat Politècnica de Catalunya
Faculty of Mathematics and Stadistics
Degree in Mathematics

in Barcelona

September 2018

Abstract

The aim of this thesis is to give an introduction to Natural Language Understanding. Many tools and language models are described along this work in order to teach a machine the ability to analyze and understand human speech. In the last chapter, there is an example of information extraction by an open source, machine learning program.

Contents

Chapter 0

Introduction

At the beginning, the main purpose of the thesis was building a chatbot to arrange meetings or math talks. We decided to work with RASA, an open source software that helps to develop any intelligent assistent. RASA basically is a combination of Rasa NLU and Rasa Core. Rasa NLU is the one in charge of interpreting messages, and Rasa Core's job is to decide what should happen next.

In this way we learned the basic notions of what the software was asking, creating a training dataset and a list of actions that the chatbot could execute.

For some reason we don't know, the two sides (NLU and Core) were outdated and when we tried trivial examples we didn't get a reliable answer. For example, a mistake we often encountered was that Rasa NLU did bad information extraction. Perhaps in the phrase *We want to eat English food*, the assistant understood that we really wanted to eat but in England. He didn't detect the named entities correctly. We noticed that RASA was updating its contents quite frequently so we moved on to something else.

So, how does a chatbot really work? In what theory is it based? The solution I ended up finding was writing a more theoretical than practical work. Along this process, I have learned which language models take part of NLU, and how many different, tedious tasks are needed to reach just one goal. I have noticed that the depth of the chosen subject is very, very large and even today Natural Language Processing is in full research. I really liked how mathematics could model something so abstract like language.

In 1950, Alan Turing published his famous paper titled "Computing Machinery and Intelligence". This paper proposed an experiment that became known as the Turing test, an attempt to determine if a machine was artificially intelligent. Basically, Turing said that if a machine could have a conversation with a human and trick the human into thinking the machine was a person itself, then it was artificially intelligent.

Over the past few decades **Artificial Intelligence** (AI) has become

very popular due to the big technological improvements. Nowadays, AI is present in so many fields and one of the most promising, useful applications is **Natural Language Processing** (NLP).

NLP is defined as the ability of a machine to analyze, understand, and generate human speech. The aim of NLP is to make interactions between computers and humans feel exactly as interactions between humans. With NLP, a computer is able to listen to a natural language being spoken by a person, understand the meaning of it, and then, if needed, respond to it by generating natural language to communicate back to the person. The most common applications for NLP today are spam filters, answering questions and summarizing information.

For example, ELIZA is an early natural language processing system created by Joseph Weizenbaum in 1966. This computer program simulated conversation by using a "pattern matching" and substitution methodology that gave users an illusion of understanding. ELIZA was one of the first chatbots of NLP softwares and one of the first programs capable of attempting the Turing Test.

So how does NLP work? In order to understand how it works, we have to differentiate between the two main components of NLP: **Natural Language Understanding** (NLU) and **Natural Language Generation** (NLG).

First, the computer must take natural language and convert it into artificial language. This is what speech recognition, or speech-to-text, does. This is the first step of NLU. Once the information is in text form, NLU can take place to try to understand the meaning of that text.

Most speech recognition systems today are based on Hidden Markov Models or HMMs. These are statistical models that turn your speech into text by making mathematical calculations to determine what you said. HMMs do this by listening to you speak, breaking it down into small units, then comparing it to prerecorded speech to determine the phoneme you said in each unit of your speech. Then, it looks at the series of phonemes and statistically determines the most likely words and sentences you were saying. It outputs this information in the form of text. Using a lexicon and a set of grammar rules, a NLG system can form complete sentences.

We could have gone into so much more detail on everything above, but this should give you a general understanding on how NLP works today.

In this thesis we will see the main steps involved in NLU and we will try to develop our own chatbot.

Chapter 1

Regular Expressions and Text Normalization

We are going to start with an important tool for expressing text patterns: the Regular Expression. Regular Expressions can be used to specify strings we might want to search from a document.

Firstly, what we are going to do with language relies on separating out or tokenizing words from running text: the task of tokenization. English words are often separated by space but sometimes this is not sufficient. For example, *New York* and *Rock & Roll* are treated as large words despite the fact that they contain white spaces. On the other hand, the string *I'm* should be treated as *I am* instead of one string.

The task of deciding if two words have the same root despite their surface differences is called **lemmatization**, another part of Text Normalization. For instance, the words *sang*, *sung* and *sings* are forms of the verb *sing*. The word *sing* is the common lemma of these words and a **lemmatizer** maps all of these to *sing*. **Stemming** is simpler than lemmatization, which consists in stripping sufixes or prefixes from the word. Text Normalization also includes sentence segmentations using cues like periods or exclamations marks.

Text Normalization means converting text to a more convenient, standard form to facilitate the manipulation of the information later on.

Finally, we have to compare words and strings. We will briefly introduce a metric called **edit distance** at the end of this chapter. This distance measures how similar two strings are based on the number of edits or changes (insertions, deletions, substitutions) are needed to change one string into another. Edit distance is an algorithm with so many applications throughout language processing, from spelling to speech recognition.

1.1 Regular Expressions

A regular expression is an algebraic notation for defining a set of strings. They are pretty useful for searching in texts and returning all matches it contains the corpus. The corpus can be a single document or a collection.

1.2 Basic regular expression patterns

The simplest kind of **Regular Expression** is a sequence of simple characters. RE stands for Regular Expression.

RE	Match
`/dance/`	dance
`/Whaaat/`	Whaaat

Regular Expressions are case sensitive; lower case `/s/` is distinct from upper case `/S/`. The right way to express it is using square brackets. The string of characters inside the brackets specifies a set of characters to match.

RE	Match
`/[pP]encilcase/`	pencilcase or Pencilcase
`/[abc]/`	'a', 'b' or 'c'
`/[1234567890]/`	Any digit

If there is a well defined sequence associated with a set of characters, the brackets can be replaced by a dash (-).

RE	Match
`/[0-9]/`	A single digit
`/[A-Z]/`	An upper case letter
`/[b-g]/`	'b', 'c', 'd', 'e', 'f' or 'g'

The square brackets can also be used to specify what a single character cannot be, by using the caret `^`. If the caret `^` is the first symbol after the open square bracket `[`, the resulting pattern is negated.

Equally important, there are more tools to express different ideas. Let's see the most common ones:

RE	Match
`/[^A-Z]/`	Not an upper case
`/[^Aa]/`	Neither 'A' nor 'a'
`/computers?/`	computer or computers
`/colou?r/`	color or colour
`/beg.n/`	The period matches any single character (begin, beg'n, begun)
`/cat|dog/`	cat or dog
`/fl(y|ies)/`	fly or flies
`/Column␣[0-9]+␣*/`	Column 1, Column 2 or Column 33, etc.
`/the*/`	thee...e (also th)

A more complex example would be a regular expression for prices:

RE	Match
`/\b[0-9]+(\.[0-9][0-9])?\b/`	$199 or $199.99
`/\b[0-9]+(\.[0-9]+)?␣*(GB|[Gg]igabytes?)\b/`	5 GB

For the purpose of simplifying regular expressions, it exists aliases for common sets of characters.

1.3 Words and Corpora

How do we decide what counts as a word? How many words are there in English? To answer these questions we need to distinguish two ways of talking about words. Types are the number of distinct words in a **corpus** (plural: **corpora**). Let V the set of words in the vocabulary and let N the total number of running words. The corpus is a computer-readable collection of text or speech.

In fact, the relationship between the number of types V and number of tokens N is called *Herdan's Law* or *Heaps' Law*.

k and $0 < \beta < 1$ are positive constants that depends on the corpus length.

$$V = kN^{\beta} \tag{1.1}$$

In section *1.5 Lemmatization and Stemming* we will talk about how to determine that two words have the same root or not. It is vital in the task of determining V.

1.4 Text Normalization

The text has to be normalized before any natural language process could take place. Three tasks are commonly applied to do it:

1. Tokenizing words from running text
2. Normalizing word-formats
3. Segmenting sentences in running text.

Let's start with an easy version of word tokenization and normalization that can be accomplished solely in a single UNIX command line. Even though, more sophisticated algorithms are generally necessary for tokenization and normalization. We will use some UNIX commands:

`tr`: used to systematically change particular characters in the input.

`sort`: which sorts input lines in alphabetical order.

`uniq`: which collapses and counts adjacent identical lines.

We can use `tr` to tokenize the words by changing every sequence of non-alphabetic characters to a newline ('`A-Za-z`' means alphabetic, the `-c` option complements to non-alphabet, and the `-s` option squeezes all sequences into a single character). Given a particlar corpus, let's practice these commands and save it into `sh.txt`. The command is

```
tr -sc 'A-Za-z' '\n' < sh.txt
```

The output is one word per line, so we can sort the lines, and pass them to `uniq -c` which collapse and count them. Finally, we can treat all upper-case to lower-case. The command is

```
tr -sc 'A-Za-z' '\n' < sh.txt | tr A-Z a-z | sort | uniq -c
```

Now we can sort again to find the most frequent words.

```
tr -sc 'A-Za-z' '\n' < sh.txt | tr A-Z a-z | sort | uniq -c | sort -n -r
```

The output shows the most frequent words in the corpus.

```
27378 the
26084 and
22538 i
19771 to
17481 of
14725 a
13826 you
12489 my
11318 that
11112 in
...
```

While the Unix command sequence just removed all the numbers and punctuation, for most NLP applications we'll need to keep these in our tokenization. We often want to treat punctuation as a separate token; commas are a useful piece of information for parsers, periods help indicate sentence boundaries. But we'll often want to keep the punctuation that occurs word internally, in examples like m.p.h, AT&T, cap'n. Special characters and numbers will need to be kept in prices ($45.55) and dates (01/02/06); we don't want to segment that price into separate tokens of "45" and "55". And there are URLs (http://www.upc.edu), Twitter hashtags (#RealMadrid), or email addresses (someone@upc.edu).

1.5 Lemmatization and stemming

Lemmatization is the task of determining that two words have the same root, despite their surface differences. The words `am`, `are`, and `is` have the shared lemma `be`; the words `dinner` and `dinners` both have the lemma `dinner`. Representing a word by its lemma is important for web search, since we want to find pages mentioning `rules` if we search for `rule`. The lemmatized form of the sentence *He is reading detective stories* would be *He be read detective story*.

How is lemmatization done? The most sophisticated methods for lemmatization involve complete morphological parsing of the word. Morphology is the study of the way words are built up from smaller meaningful units called morphemes. Two broad classes of morphemes can be distinguished: **stems** (the central morpheme of the word, supplying the main meaning) and **affixes** (adding "additional" meanings of various kinds). **Stemming** consists in chopping off affixes and one of the most widely used stemming algorithms in English is the simple and efficient **Porter algorithm**.

1.6 Minimum distance

Much of natural language processing is concerned with measuring how similar two strings are. For instance, if the user typed some erroneous string and we want to know what string the user meant. **Coreference** is another remarkable example, the task of whether two strings refers to the same entity.

Edit distance gives us a way to quantify both of these intuitions about string similarity. More formally, the minimum edit distance between two strings is defined as the minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another. There are several distances in the sense that we can assign particular costs to editing operations. The algorithm that resolves the minimum edit distance between two words is based on dynamic programming.

Chapter 2

Language Modeling with N-grams

In this chapter we will formalize the intuition of assigning a probability to each possible next word. The same models will also serve to assign a probability to an entire sentence. Models that assign probabilities to sequences of words are called language models or LMs. We will see the simplest model, the N-gram. An N-gram is a sequence of N words: a 2-gram (or bigram) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (or trigram) is a three-word sequence of words like "please turn your", or "turn your homework". We'll see how to use N-gram models to estimate the probability of the last word of an N-gram given the previous words, and also to assign probabilities to entire sequences.

Whether estimating probabilities of next words or of whole sequences, the N- gram model is one of the most important tools in speech and language processing.

Probabilities are essential in any task in which we have to identify words in noisy, ambiguous input, like speech recognition or handwriting recognition. Assigning probabilities to sequences of words is also essential in machine translation.

2.1 N-grams

Let's begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is *"its water is so"* and we want to know the probability that the next word is *transparent*:

$$P(transparent|its\ water\ is\ so) = \frac{C(its\ water\ is\ so\ transparent)}{C(its\ water\ is\ so)} \tag{2.1}$$

Given a corpus, one possible way to estimate this probability is to count how many times the history h was followed by the word w, *C(hw)*.

This method works fine in many cases but it turns out that the data isn't big enough to give us good estimations in most cases. It depends on the size of the corpus. For this reason, we need cleverer ways to deal with this problem. Let's introduce some notation. To represent the probability of a particular random variable X_i taking the value *"the"*, P(X_i = *the*), we will use P(*the*). We will represent a sequence of words $w_1...w_n$ as w_1^n. So, applying the chain rule of probability we get this expression:

$$P(w_1^n) = P(w_1)P(w_2|w_1)P(w_3|w_1^2)...P(w_n|w_1^{n-1}) = \prod_{k=1}^{n} P(w_k|w_1^{k-1}) \quad (2.2)$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. Otherwise, we don't know to calculate the exact probability of a word given a long sequence of preceding words. So what we are going to do is approximating the history by just the last few words.

When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1}) \quad (2.3)$$

The Markov assumption asserts that the probability of a word depends only on the previous word. **Markov models** are the kind of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past.

Given the bigram assumption, the resulting probability of a complete word sentence is

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \quad (2.4)$$

We get the Maximum Likelihood Estimation (MLE) for the parameters of an N-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1. In a bigram model we get this expression

$$P(w_k|w_{k-1}) = \frac{C(w_{k-1}w_k)}{C(w_{k-1})} \quad (2.5)$$

Let's move to an example. We will compute the probability of a word sentence using a bigram model. The sentence will be *I want English food*. However, we need to extract some information before anything else. First, we count how many words are in the given corpus, the Berkeley Restaurant Project corpus.

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

The following chart shows the bigram probabilities for the previous sentence. Each cell is calculated using the equation (2.5).

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Table 2.1: Example

Now if we want to compute the sentence *I want English food* we need the following computations, where `<s>` and `<\s>` corresponds to the beginning and final of the sentence

$$P(\texttt{i}|\texttt{<s>}) = 0.25 \qquad P(\texttt{english}|\texttt{want}) = 0.0011$$
$$P(\texttt{food}|\texttt{english}) = 0.5 \qquad P(\texttt{<\s>}|\texttt{food}) = 0.68$$

Finally we can compute the desired probability just multiplying the appropriate bigram probabilities together:

$$P(\texttt{<s> i want english food <\s>}) = P(\texttt{i}|\texttt{<s>})\texttt{P(want|i)...P(<\s>|food)} =$$

$$= 0.25 \times 0.33 \times .0011 \times 0.5 \times 0.68 = 0.000031$$

In practice it is more common to use trigram instead of bigram models.

2.2 Language Models

Given two probabilistic models, the better model is the one that better predicts the details of the test data, therefore it will assign a higher probability to the test data.

The N-gram model, like many statistical models, is dependent on the training corpus. The higher value of N the better job of modeling the training corpus.

Chapter 3

Neural Networks

In this chapter we will talk about neural networks and their applications as classifiers. Neural networks are a very important tool for language processing. Their origins come from a simplified model of the human neuron which can be described in terms of propositional logic.

So let's begin with the building block of a neural network, the unit.

3.1 Units

A neural network is made by units. A unit takes a set of real valued numbers as input and produces an output. In fact, a neural unit is taking a weighted sum of its inputs $x_1, x_2, ..., x_n$ and one additional term called **bias term** b. So the weighted sum z can be represented as

$$z = b + \sum_i w_i x_i \tag{3.1}$$

or more convenient using vector notation

$$z = w \cdot x + b \tag{3.2}$$

We will introduce three non-linear functions f that neural units apply to z. The *sigmoid* (3.3), the *tanh* (3.4) and the *rectified linear unit* or *ReLU* (3.5). Each one has different properties that make them useful for different language applications or network architectures.

$$y = \frac{1}{1 + e^{-z}} \tag{3.3}$$

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.4}$$

$$y = \max(z, 0) \tag{3.5}$$

3.2 The XOR problem

The power of neural networks relies on combining these units into larger networks. It is proved that a single neural unit cannot compute some very simple functions of its input. Let's try to compute simple logical functions of two inputs like `AND`, `OR` and `XOR`. The **perceptron** is a very simple neural unit that has a binary output and no non-linear activation function f. The **perceptron** is

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases} \tag{3.6}$$

and the truth tables for the logical functions are

	AND			OR			XOR	
$x1$	$x2$	y	$x1$	$x2$	y	$x1$	$x2$	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

For instance, for the logical function `AND`, a possible set of weights and bias could be $w_1 = 1, w_2 = 1$ and bias $b = -1$. On the other hand, for the function `OR` could be $w_1 = 1, w_2 = 1$ and bias $b = 0$.

However, it turns out that it is not possible to build a **perceptron** to compute the logical `XOR`. The reason behind this fact is that the **perceptron** is a linear classifier. For two dimensional input $x1$ and $x2$, the equation relate them in form of a line. This line acts as a decision boundary in two dimensional space.

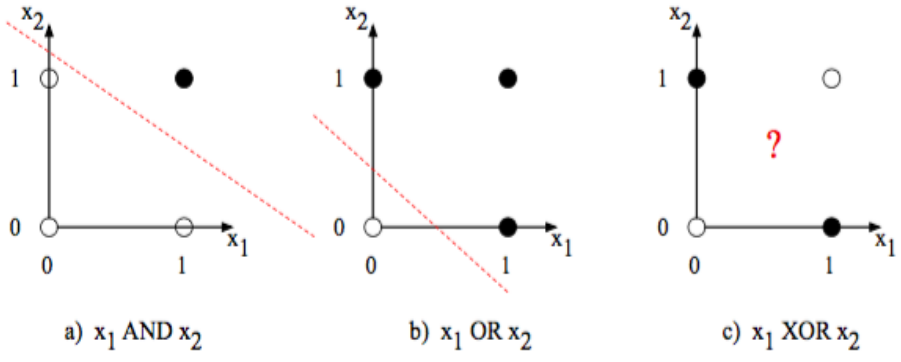

Figure 3.1: The functions AND, OR and XOR represented with the perceptron. Filled circles mean output 1 and white ones output 0.

Goodfellow et al. (2016) gave a solution that computes XOR using two

layers of ReLU-based units. They made it by merging the two input points $x = [0\ 1]$ and $x = [1\ 0]$ into only one $h = [0\ 1]$. In that case, the merger makes it easy to linearly separate the positive and negative cases of `XOR`.

3.3 Feed-Forward Neural Networks

A feed-forward neural network is a multilayer network that satisfies the following properties: the units are connected with no cycles, the outputs from units in each layer are passed to units in the higher layer, and no outputs are passed back to lower layers.

In a simple feed-forward networks we have three layers (input, hidden and output layer) and three types of nodes (input, hidden, and output units. The core of the neural network is the hidden layer formed of hidden units. Normally, each layer is fully-connected, meaning that each unit in each layer takes as input the outputs from all the units in the previous layer.

Let's start with some notation for this neural network. We will have a matrix W and a bias vector b for the entire hidden layer. Each element W_{ij} represents the weight of the connection from the input unit x_i to the hidden unit h_j. As a result, the hidden layer computation for a feed-forward network is easier:

$$h = f(Wx + b) \tag{3.7}$$

Let's assume that f is the sigmoid function we have seen before.

Generally, the goal of the network is to make some sort of classification decision, and so we will focus on the case of classification. Additionally, Part-of-Speech Tagging might have one output node for each potential category of word and the values of all the output nodes must sum one.

The output layer has also a weight matrix U. The weight matrix U is multiplied by h to get z.

$$z = Uh \tag{3.8}$$

After all, z can't be the output of the classifier, since it's a vector of real-valued numbers instead of a vector of probabilities. The softmax function will normalize the vector of real values. If z has dimension k then

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}, \quad 1 \leq i \leq k \tag{3.9}$$

An example that shows a simple feed-forward neural network could be

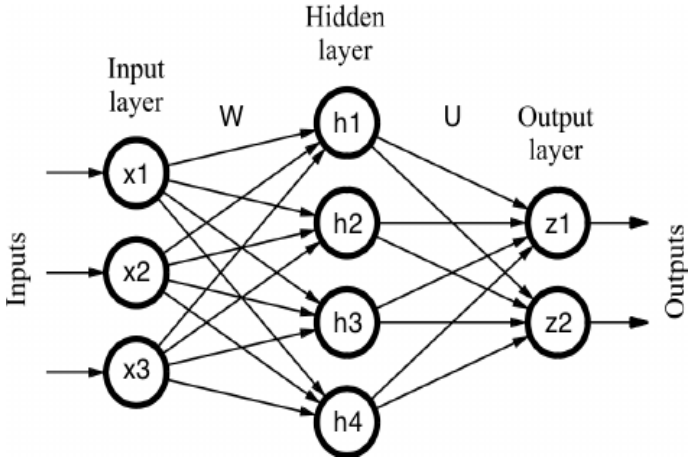

Figure 3.2: An example of a Feed-Forward Neural Network

3.4 Training Neural Nets

Training a neural net means setting the weights W and biases b for each layer using optimization methods like **stochastic gradient descent**. The intuition of this method consists in starting with some initial guess at the variable θ (which contains all the parameters from W and b for each layer) and then move slightly forward in a direction that improves our system.

We know the right answer for each observation in the training set, so our goal is to change weights in a way that improves the system. We measure the distance between the system output and the desired output. The distance is called the **loss function** and it is expressed as

$$L(f(x;\theta), y) = \text{How much } f(x) \text{ differs from the true } y \tag{3.10}$$

In this chapter we will use the negative log likelihood because it insures that as probability of the correct answer is maximized, the probability of all the incorrect answers is minimized:

$$L(\hat{y}, y) = -\log p(\hat{y}_i) \tag{3.11}$$

where $\hat{y}_i$ is the system's output of the correct class and i corresponds to the position of 1.

Given a loss function, the goal in training is to find the minimum of the loss function moving the parameters.

3.5 Neural Language Models

One application of neural network might be language modeling: predicting words from previous word context. We have seen in chapter 2 how N-grams worked, but neural net-based language models turn out to have many advantages. Let's describe a simple feed-forward neural language model.

A **feed-forward neural LM** is a standard feed-forward network that takes as input at time t a representation of some number of previous words w_{t-1}, w_{t-2}, etc and outputs a probability distribution over the possible next words. We will approximate with the N previous words

$$P(w_t|w_1^{t-1}) \approx P(w_t|w_{t-N+1}^{t-1}) \tag{3.12}$$

From now on let's say $N = 4$.

In neural language models, each word is represented as a vector of real numbers of dimension d, generally $50 < d < 500$. These vectors are called **embeddings**.

Imagine that we had an embedding dictionary E that gives us, for each word in our vocabulary V, the vector for that word.

A **one-hot vector** is a vector that has one element equal to 1 while all the other elements are set to zero.

In summary, the main steps of a feed-forward neural language model are the following:

1. Select three embeddings from E: Given the 3 previous words ($N = 4$), we create 3 one-hot vectors (`[0...0 1 0...0]`) looking at their indices in V. Then these one-hot vectors are multiplied by the embeddding matrix E, to give the projection layer.

2. Multiply by W: Concatenate the three embeddings and multiply by W (and add b) in order to get the hidden layer.

3. Multiply by U and apply softmax: h is multiplied by U and then the softmax function estimates the probability $P(w_t = i|w_{t-1}w_{t-2}w_{t-3})$ for each node. This estimation shows what probability has the word i-th in V to be w_t.

Chapter 4

Hidden Markov Models

So far we have seen two different tools for language processing, the N-gram model and the neural network. Now it's time to talk about the **hidden Markov model**, HMM, one of the most important machine learning models in speech and language processing. The HMM is a probabilistic sequence model, whose task is to assign a label or class to each unit in a sequence. Given a sequence of units (words, letters, whatever), the model compute a probability distribution over possible sequences of labels and choose the best label sequence.

Let's begin with the basic theory around HMM, the Markov Chain.

4.1 Markov Chains

A **Markov Chain** is a stochastic (random process) model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. Figure 4.1 shows an example to illustrate this idea

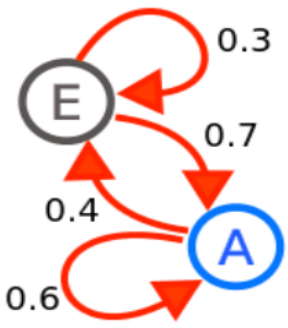

Figure 4.1: A Markov Chain

This is a representation of a two-state Markov process, with the states labelled E and A. Each number represents the probability of the Markov process changing from one state to another state, with the direction indicated by the arrow.

For example, if the Markov process is in state A, then the probability it changes to state E is 0.4, while the probability it remains in state A is 0.6.

We can check that the probabilities on all arcs leaving a state must sum 1.

Now if we represent these states as words, a Markov chain is able to assign a probability to a sequence of words $w_1...w_n$.

In a **first-order Markov chain**, the probability of a state depends only on the previous state. This is called the **Markov assumption**:

$$P(q_i|q_1...q_{i-1}) = P(q_i|q_{i-1}) \tag{4.1}$$

where q_k is the k-th state $\forall k$.

4.2 The Hidden Markov Model

A Markov chain is useful when we need to compute a probability for a sequence of events that we can observe. However, the events we are interested in may not be observable.

A HMM allows us to work with observed events (words) and hidden events (part-of-speech tags).

Let's introduce some notation to specify a HMM:

Let $Q = \{q_1, q_2...q_N\}$ be the set of N states.

Let $A = (a_{01}, a_{02}, ..., a_{n1}, ..., a_{nn})$ be the transition probability matrix, where a_{ij} stands for the probability of moving from state i to state j. Remind that $\sum_{j=1}^{n} a_{ij} = 1 \ \ \forall i$.

Let $O = o_1, o_2, ..., o_T$ be a sequence of T observations.

Let $B = b_i(o_t) \ \ \forall i, t$ be a sequence of observation likelihoods, also called emission probabilities, each expressing the probability of an observation o_t being generated from state i.

Let q_0, q_F be special states, the **start** and **end** states.

A first-order HMM satisfies the Markov assumption (4.1) and also one more, the **output independence**:

$$P(o_i|q_1...q_N, o_1...o_T) = P(o_i|q_i) \tag{4.2}$$

To illustrate these models, we'll practice with an example.

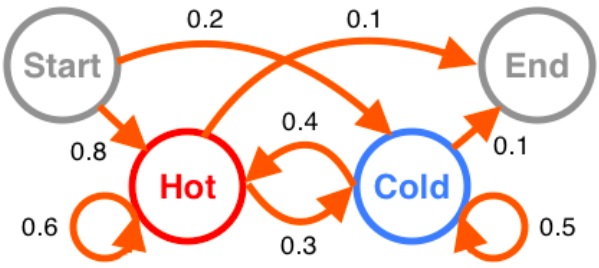

Figure 4.2: A HMM for relating the numbers of ice creams eaten by Bob

Imagine we want to know the weather of the last week. We don't know the temperature for every day (Hot or Cold) but luckily we have Bob's diary, which lists how many ice creams he ate every day. Our goal is to use these observations to estimate the temperature every day.

In essence, given a sequence of observations O, each one an integer corresponding how many ice creams eaten on a given day, figure out the correct hidden sequence Q of weather states which caused Bob to eat the ice cream.

The two hidden states are Hot and Cold weather, and the observations (let's say $V = \{1, 2, 3\}$) correspond to the number of ice cream eaten by Bob.

The emission probabilities B are

$$B_{Hot} : \begin{bmatrix} P(1|\text{Hot}) \\ P(2|\text{Hot}) \\ P(3|\text{Hot}) \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.4 \\ 0.4 \end{bmatrix} \qquad B_{Cold} : \begin{bmatrix} P(1|\text{Cold}) \\ P(2|\text{Cold}) \\ P(3|\text{Cold}) \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.4 \\ 0.1 \end{bmatrix}$$

At this point we have seen the structure of a HMM. However, we didn't talk about the algorithms for computing things with them.

In many cases, we don't gather all the components that specifies a HMM, but we can find useful solutions to them. There are three fundamental problems:

1. Likelihood Problem: Given an HMM $\lambda = (A, B)$ and an observation sequence O, determine the likelihood $P(O|\lambda)$.

2. Decoding Problem: Given an HMM $\lambda = (A, B)$ and an observation sequence O, discover the best hidden state sequence Q.

3. Learning Problem: Given an observation sequence O and the set of states in the HMM, learn the HMM parameters A and B.

We are going to develop the decoding problem in the next chapter because part-of-speech tagging is very related to decoding. We will introduce the **Viterbi algorithm** for decoding, which chooses the tag sequence that is most probable given the observation sequence of n words.

Chapter 5

Part-of-Speech Tagging

Knowing whether a word is a noun or a verb tells us a lot about neighboring words and about the syntactic structure around the word. **Parts-of-speech** (also known as syntactic categories) are useful for finding named entities like people and other information extraction tasks.

This chapter introduces the Hidden Markov Model algorithm for tagging parts-of-speech to words. However, it is worth to mention that it is not the only model that exists for that. The maximum entropy Markov Model also does this task. Firstly, we are going to see the tagset that most modern language processing uses on English, the 45-tag Penn Treebank. This tagset has been used to label a wide variety of corpora.

Tag	**Description**	**Example**	**Tag**	**Description**	**Example**
CC	coordin. conjuction	and, but	SYM	symbol	+, %, &
CD	cardinal number	one, two	TO	"to"	to
DT	determiner	a, the	UH	interjection	ah, oops
EX	existential 'there'	there	VB	verb base form	eat
FW	foreign word	mea culpa	VBD	verb past tense	ate
IN	preposition	of, in, by	VBG	verb gerund	eating
JJ	adjective	yellow	VBN	verb past participle	eaten
JJR	adj. comparative	bigger	VBP	verb non-3sg pres	eat
JJS	adj. superlative	wildest	VBZ	verb 3sg pres	eats
LS	list item marker	1,2, One	WDT	wh-determiner	which, that
MD	modal	can, should	WP	wh-pronoun	what, who
NN	noun, sing. or mass	llama	WPS	possessive wh-	whose
NNS	noun, plural	llamas	WRB	wh-adverb	how, where
NNP	proper noun sing	IBM	$	dollar sign	$
NNPS	proper noun plural	Carolinas	#	pound sign	#
PDT	predeterminer	all, both	“	left quote	‘ or “
POS	possessive ending	's	”	right quote	’ or ”

Tag	Description	Example	Tag	Description	Example
PRP	personal pronoun	I, you, he	(	left parenthesis	[,(,{,¡
PRPS	possessive pronoun	your, one's	)	right parenthesis	],),},¿
RB	adverb	quickly, never	,	comma	,
RBR	adverb comparative	faster	.	sentence-final punc	. !
RBS	adverb superlative	fastest	:	mid-sentence punc	:; ··· -
RP	particle	up, off			

Table 5.1: Penn Treebank part-of-speech tags

Corpora labeled with parts-of-speech are crucial training (and testing) sets for statistical tagging algorithms.

Three main tagged corpora are consistently used for training and testing part-of-speech taggers for English: the Brown, the Wall Street Journal and the Switchboard corpus. All these corpora are hand-corrected by human annotators.

Tagging is a disambiguation task because words are ambiguous. The goal is to find the correct tag for the situation. A HMM for Part-of-Speech Tagging seek the higher tag accuracy and now we are going to describe it in the following section.

5.1 HMM Part-of-Speech Tagging

HMMs for part-of-speech tagging are trained on a fully labeled dataset, setting parameters by maximum likelihood. In the first place, let's start with some intuition of HMM decoding.

The tag sequence that is most probable given the observation sequence of n words w_1^n is

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n|w_1^n) \tag{5.1}$$

by using the Bayes' rule

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} \frac{P(w_1^n|t_1^n)P(t_1^n)}{P(w_1^n)} \tag{5.2}$$

and simplifying the denominator.

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(w_1^n|t_1^n)P(t_1^n) \tag{5.3}$$

HMM taggers make two simplifying assumptions.

$$P(w_1^n|t_1^n) \approx \prod_{i=1}^{n} P(w_i|t_i) \tag{5.4}$$

$$P(t_1^n) \approx \prod_{i=1}^{n} P(t_i|t_{i-1}) \tag{5.5}$$

Equation (5.4) explains the dependency of a word from neighboring words and tags, and equation (5.5) is the bigram assumption.

In summary, plugging previous equations we have

$$\hat{t}_1^n = \underset{t_1^n}{\text{argmax}} = \underset{t_1^n}{\text{argmax}} \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1}) \tag{5.6}$$

where $P(w_i|t_i)$ and $P(t_i|t_{i-1})$ are called **emission** and **transition** probabilities, respectively. These are estimated using equation (2.5) from chapter 2, counting how often the first element is followed by the second.

5.2 A decoding example

Let's compute the best sequence of tags that corresponds to the following sequence of words: *Janet will back the bill.* After training, we compute the transition probability matrix A and the observation likelihoods $b_i(o_t)$.

	NNP	**MD**	**VB**	**JJ**	**NN**	**RB**	**DT**
$< s >$	0.2767	0.0006	0.0031	0.0453	0.0449	0.0510	0.2026
NNP	0.3777	0.0110	0.0009	0.0084	0.0584	0.0090	0.0025
MD	0.0008	0.0002	0.7968	0.0005	0.0008	0.1698	0.0041
VB	0.0322	0.0005	0.0050	0.0837	0.0615	0.0514	0.2231
JJ	0.0366	0.0004	0.0001	0.0733	0.4509	0.0036	0.0036
NN	0.0096	0.0176	0.0014	0.0086	0.1216	0.0177	0.0068
RB	0.0068	0.0102	0.1011	0.1012	0.0120	0.0728	0.0479
DT	0.1147	0.0021	0.0002	0.2157	0.4744	0.0102	0.0017

Table 5.2: The A transition probabilities $P(t_i|t_{i-1})$ computed from WSJ corpus.

	Janet	**will**	**back**	**the**	**bill**
NNP	0.000032	0	0	0.000048	0
MD	0	0.308431	0	0	0
VB	0	0.000028	0.000672	0	0.000028
JJ	0	0	0.000340	0.000097	0
NN	0	0.0002000	0.000223	0.000006	0.002337
RB	0	0	0.010446	0	0
DT	0	0	0	0.506099	0

Table 5.3: Observation likelihoods B computed from WSJ corpus.

The sentence *Janet will back the bill* contain words with ambiguities. For instance, the word *will* can appear as VB, MD or NN. The **Viterbi algorithm** finds the optimal sequence of tags and is able to resolve this ambiguity. Given an observation sequence and an HMM $\lambda = (A, B)$, it returns the state path through the HMM that assigns maximum likelihood to the observation sequence. The code can be found at wikipedia.

Ultimately, the correct tagging provided by this method is

Janet will back the bill.
NNP MD VB DT NN

It is worth to mention that HMM algorithms generally are extended to trigram assumptions.

Chapter 6

Information Extraction

The goal of **information extraction**, IE, is turning the unstructured information embedded in texts into structured data.

The first task in most IE tasks is to find the **named entities**, known as named entity recognition (NER). A named entity type might be people, places, organizations, etc. Then those entities are classified into types. Named entities are useful for many other language processing tasks like question answering.

In NER there is also ambiguity. For example, the word *Washington* is classified in different named entities: person, location, political entity, organization and vehicle. For this reason, sequence labeling techniques are needed for this task, specially Conditional Random Fields (CRFs). It is true that HHMs are also sequence tagging models but CRF models work better for this task. In fact, a linear-chain CRF, can be thought as an undirected graphical model version of HMM.

6.1 The CRF Model

Let $x_{1:N}$ be the observations (words in a document) and $z_{1:N}$ the hidden labels (tags). A linear chain CRF defines a conditional probability as

$$P(z_1^N|x_1^N) = \frac{1}{Z} \exp\left(\sum_{n=1}^{N} \sum_{i=1}^{Z} \lambda_i f_i(z_{n-1}, z_n, x_1^N, n) \right) \tag{6.1}$$

The scalar Z is a normalization factor to make $P(z_{1:N}|x_{1:N})$ a valid probability over label sequences. It is defined as

$$Z = \sum_{z_1^N} \exp\left(\sum_{n=1}^{N} \sum_{i=1}^{Z} \lambda_i f_i(z_{n-1}, z_n, x_1^N, n) \right) \tag{6.2}$$

We sum over $n = 1...N$ word positions in the sequence. For each position, we sum over $i = 1...F$ weighted features. The scalar λ_i is the weight for **feature**

functions f_i. The λ_i parameters of the CRF model must be learned, similar to HMMs paramenters (A, B).

6.2 Feature Functions f_i

The feature functions are key in CRFs. The general form of a feature function is $f_i(z_{n-1}, z_n, x_{1:N}, n)$, which looks at the pair of adjacent states z_i, z_{i-1}, the whole input sequence $x_{1:N}$ and where we are in the sequence. For example, it could be a binary function like

$$f_1(z_{n-1}, z_n, x_{1:N}, n) = \begin{cases} 1 & \text{if } z_n = \text{PERSON and } x_n = \text{John} \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

This feature depends on λ_1. If $\lambda_1 > 0$, it increases the probability of the tag sequence $z_{1:N}$. On the other hand, if $\lambda_1 < 0$ the CRF model will try to avoid the tag PERSON for JOHN. When $\lambda_1 = 0$ the feature has no effect. The weight λ_1 may be set by domain knowledge or be learned from corpus.

Another example of feature could be

$$f_2(z_{n-1}, z_n, x_{1:N}, n) = \begin{cases} 1 & \text{if } z_n = \text{PERSON and } x_{n+1} = \text{said} \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

where this feature is active if the current tag PERSON and the next word is *said*. Moreover, note f_1 and f_2 can be both active which is called an example of overlapping features. Features are not limited to binary functions, any real-valued one is allowed.

6.3 CRF training

Training means finding the λ parameters in a CRF. For this task we need a fully labeled data sequences $\{(x^{(1)}, z^{(1)})...(x^{(m)}, z^{(m)})\}$ as training data, where $x^{(i)} = x_{1:N_i}$ is the *i-th* observation sequence $\forall i$. The objective for parameter learning is to maximize the conditional likelihood of the training data

$$\sum_{j=1}^{m} \log P(z^{(j)}|x^{(j)}) \tag{6.5}$$

The standard approach is to compute the gradient of the objective function and use it in an optimization algorithm.

Now we are going to move to a more practical point of view. Let's see the role of Natural Language Understanding using information extraction.

Chapter 7

Snips NLU

Snips NLU is a Natural Language Understanding Python library that allows to parse sentences written in natural language, and extract structured information.

The Snips NLU engine, in its default configuration, needs to be trained on some data before it can start extracting information. Thus, the first thing to do is to build a dataset that can be fed into Snips NLU.

We will work on a example to illustrate the main purpose.

As we said, Snips NLU need to be trained. The dataset can be found at `http://github.com/snipsco/snips-nlu/blob/master/snips_nlu_samples/sample_dataset.json`

Once Snips NLU is installed in the computer, the following lines

```
import io
import json
with io.open("path/to/sample_dataset.json") as f:
    sample_dataset = json.load(f)

from snips_nlu import load_resources, SnipsNLUEngine

load_resources(u"en")
nlu_engine = SnipsNLUEngine()

nlu_engine.fit(sample_dataset)

parsing = nlu_engine.parse(u"What will be the weather in San Francisco next week?")
print(json.dumps(parsing, indent=2))
```

leads to the structured data

```
{
  "input": "What will be the weather in San Francisco next week?",
  "intent": {
    "intentName": "sampleGetWeather",
    "probability": 0.641227710154331
  },
  "slots": [
    {
      "range": {
        "start": 28,
        "end": 41
      },
      "rawValue": "San Francisco",
      "value": {
        "kind": "Custom",
        "value": "San Francisco"
      },
      "entity": "location",
      "slotName": "weatherLocation"
    },
    {
      "range": {
        "start": 42,
        "end": 51
      },
      "rawValue": "next week",
      "value": {
        "type": "value",
        "grain": "week",
        "precision": "exact",
        "latent": false,
        "value": "2018-02-12 00:00:00 +01:00"
      },
      "entity": "snips/datetime",
      "slotName": "weatherDate"
    }
  ]
}
```


In the context of information extraction, an **intent** corresponds to the action or intention contained in the user's query, which can be more or less explicit.

Chapter 8

Conclusions

The purpose of this work has been to give an introduction on how a machine could understand human speech.

Natural Language is very complex and ambiguous. Therefore, it must be studied thoroughly at different levels, both semantically and structurally. I have learned different tools and linguistic models that use statistical techniques to give computer systems the ability to learn with data. For example, this technique, known as machine learning, is applied to neural networks and HMM, seen in the thesis.

This work has also taught me the theoretical depth that these mathematical models can achieve and many clever ways to deal with some language problems.

Bibliography

[1] Bates, Madeleine, *Models of natural language understanding*, Vol. 92, Cambridge, October 1995.

[2] Daniel Jurafsky and James H. Martin, *Speech and Language Processing*, 3rd ed., Pearson Prentice Hall, 2017.

[3] Christopher D. Manning and Hinrich Schütze, *Foundations of Statistical Natural Language Processing*, The MIT Press, 1999.

[4] Russell, Stuart J. and Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2003, ISBN 0-13-790395-2.

[5] Adrien Ball and Clement Doumouro, *Snips Natural Language Understanding*, 2018.

[6] Xiaojin Zhu, *Conditional Random Fields*, CS769 Spring 2010 Advanced Natural Language Processing.