Chapter 1

Introduction

1.1 Applications of NLP

Natural language processing (NLP) is about making computers do all kinds of things with natural language (that is, human languages, like English or Chinese). I can think of three broad areas where this would be useful.

First, we'd like to be able to interact with computers using natural language. This idea has captured imaginations for a long time, since at least *Star Trek* and *2001: A Space Odyssey*'s HAL 9000, and became a major goal of NLP research and development – for example, in the 1990s Bill Gates was a major advocate, saying things like "Most of [our research] now is focused on what we call the natural interface – the computer being able to listen and talk and recognize handwriting. . . .Now we're betting the company on these natural interface technologies." [1] Today, the technology for this is good enough for systems like Apple Siri to become commercial products, but if you've used such systems, you know that they still have a long way to go.

There are also situations where we'd like to understand or communicate with other humans, but face a limitation that we'd like NLP to overcome. One limitation is when the other person doesn't speak the same language, and we'd like to use NLP to translate between the two languages. Historically, this was the oldest application of NLP, and indeed one of the very oldest applications of computers. The most well-known early system was developed by Georgetown and IBM in the early 1950s for translating Russian into English. Now, you can use Google Translate to get translations that are very high quality under the right conditions, but still need work under other conditions (like, translating Shakespeare into Japanese).

Another limitation is when there is too much language: I can read a book, but I can't read a million books. I'd like to use NLP to read them for me and then answer questions about them, summarize them, extract relevant pieces of information from them, and so on. As more and more data comes into existence, and much of it in the form of natural language, this application of NLP has become more and more important. One high-profile recent demonstration of this use of NLP was IBM's Watson, which defeated Ken Jennings at Jeopardy in 2011. But Watson's spectacular failure in the final round of this match showed, again, that

[1] Remarks at Gartner Symposium, 1997/10/06, Orlando, FL.

there's still a long way to go.

1.2 Stages of NLP

How can we make computers do these things? We can break a typical NLP system into several stages of processing. These stages will also form the main units of the course (after an initial introduction focusing on machine translation).

1.2.1 Text

Raw language input exists in many forms: primarily, speech (for spoken languages) and signing (for sign languages), and secondarily, all the ways that people have come up with over the centuries for encoding language, like handwriting, printing, and keyboard input. (There are other forms of language like whistling and drums that are not the focus of any serious NLP research that I'm aware of.)

The first stage of language processing involves ingesting language in one or more of the above forms and getting it into a representation that computers can do useful things with. Nearly always, that representation is plain text. Converting each of the above forms of language into text is a research field in its own right: speech recognition, sign language recognition, handwriting recognition, optical character recognition. Even converting typing into text is not trivial (think about mobile devices, or users with disabilities), and research on text input methods sits at the border between human-computer interaction and NLP. In this part of the course, we'll take a brief look at these topics and how they interface with natural language processing.

1.2.2 Structure

In the next part of the course, we'll study how computers can automatically discern the *structure* of natural language text (Figure 1.1): words combine to form phrases, phrases combine to form sentences; going in the other direction, words can often be broken down into smaller units called *morphemes*.

The reason that we're interested in structure is that we believe that structure is the key to understanding language, as well as other understanding-like tasks. For example, suppose you want to translate this sentence into Latin:

(1.1)	spiritus	nobile	minimum	virum	auget
	spirit	noble	smallest	man	embiggens

In order to do this right, your system has to learn that in Latin, verbs (auget = embiggen) usually come after their objects (minimum virum = the smallest man). These elements belong to syntactic structure, which is not explicit in our data (punctuation in text and intonation in speech give hints, but not very much).

The big problem at this stage is *ambiguity*: a given expression can have more than one structure. In fact, most expressions have many, many structures. So the computer's job is to figure out which structure out of all the possible structures is the right one.

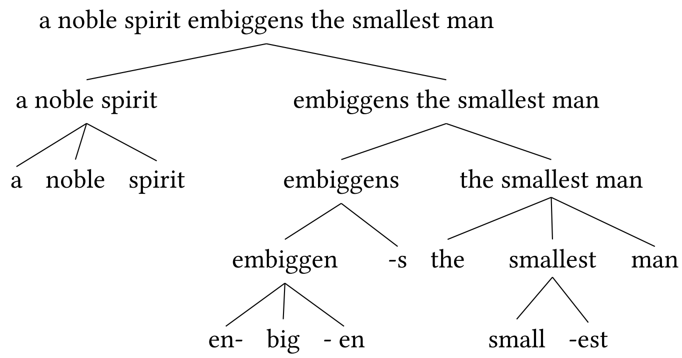

Figure 1.1: Levels of structure.

1.2.3 Meaning

Although some applications (like grammar checking) might stop at analyzing structure, most interesting applications of NLP do something with the meaning of natural language input.

The principle of *compositionality*, which originates in the philosophy of language, says that the meaning of an expression is a function of the meanings of its subexpressions. A sentence's meaning is a function of its phrases' meanings, a phrase's meaning is a function of its words' meanings, and a word's meaning is a function of its morphemes' meanings. So we use the structure produced in the previous stage (1.2.2), each level of which has meaning (as opposed to the first stage (1.2.1): a letter/sound *r* doesn't have meaning, but a morpheme *re-* does).

So, having determined the structure of a piece of text, computing its meaning is thought to be a bottom-up process, from the morphemes at the bottom all the way up to sentences and beyond.

1.2.4 Generation

Finally, in some applications, we need the computer to go in the reverse direction, from internal representations of meaning to spoken or written language. In the last (shortest) part of the course, we'll talk about methods for doing this.

1.3 Approaches to NLP

1.3.1 Linguistics

The above division of natural language processing into stages is informed by the science of linguistics. Very early NLP was more or less *ad hoc*, but in the 1960s, a committee of scientists appointed by the government prescribed more basic research into *computational linguistics*, the use of computational methods for the scientific study of language. The hope was, and is, that by understanding better how human language works, we will do a better job programming computers to imitate it.

For some (myself included), computational linguistics is interesting even if it doesn't lead to NLP applications. Although human languages seem so different from the formal languages and computer languages invented by people, they, too, are governed by rules, rules that you were never explicitly taught by your parents or in school. To take one example, if you are a native speaker of English, then you know that the sentence

(1.2) Who did Bill ask when arrived?

is not English. You have to say "Bill asked when *who* arrived?" instead. A theory of language should explain why, and one particularly simple explanation comes from computational linguistics, using a grammar slightly more powerful than a context-free grammar, called a tree-adjoining grammar.

1.3.2 Learning

In the 1990s, there was a second major shift in the way natural language processing was done. Instead of just building systems that simulate human use of language, we began trying to simulate a second human behavior: *learning* language. In other words, we used to program the rules of language directly into the computer, but now we program computers to learn the rules, and their weights, automatically from *data*. So the goal of modern, statistical NLP is to build computer systems that learn from data how to use human language.

Initially, people who used linguistics and the people who used statistics were at odds with each other. The reason was simple: linguistics is primarily interested in structures and representations that exist in the mind and cannot be directly observed, whereas statistics are based on observable quantities. So for a while, it was assumed that if you were using linguistics, you did not believe in statistics, and if you were using statistics, you did not believe in linguistics.

1.3.3 Linguistics and Learning

Over time, however, a synthesis emerged. First, people started to build datasets annotated with linguistic structures (for example, the Penn Treebank), thus making unobservable structures and representations observable. Thus it became possible to use statistics and linguistics together: "linguistics tells us what to count, and statistics tell us how to count it" (Joshi).

Second, people started to develop models that can learn unobserved things (for example, syntactic structure). These models, though not tied to a particular linguistic theory, were nevertheless informed by what linguistics says about how language works (for example, syntactic structure is recursive, so our models should be recursively structured as well).

There is a third possible outcome for the role of linguistics in NLP. It may end up that we do not need linguistics either for creating datasets or for designing models, but that vanilla models will successfully learn their own representations oflanguages. In that case, the role of linguistics will be to analyze and explain what computers learn about language (just as it now tries to analyze and explain what humans learn about language).

I don't know which of these outcomes, or which combination of these outcomes, will prevail, but I think it must be some combination of these, and so there will always be a role for linguistics in NLP. Consequently, although we will try to cover state-of-the-art machine learning methods for NLP, there's a significant emphasis on linguistics and older NLP methods as well.

1.4 Preliminaries

1.4.1 Probability

Below is a very brief review of basic probability theory. The notation used for probabilities in NLP is a little sloppy, but hopefully this is good enough. For a proper treatment, see the textbook by Bertsekas and Tsitsiklis (2008).

Random variables. A random variable is a variable with a different random value in each "experiment". For example, if our experiments are coin flips, we could define a random variable $C \in \{\text{heads}, \text{tails}\}$ for the result of the flip. Or, if our experiments are the words of a speech, we could define a random variable $W \in \{\text{a}, \text{aa}, \text{ab}, \ldots\}$ for the words spoken. If X is a random variable with values in $\mathcal{X}$, we call $P(X)$ the distribution of X. If $x \in \mathcal{X}$, we write $P(X = x)$ for the probability that X has value x. We must have

$$\sum_{x \in \mathcal{X}} P(X = x) = 1.$$

For example, if $P(W)$ is a distribution over English words, we might have

$$\begin{aligned} P(W = \text{the}) &= 0.1 \\ P(W = \text{syzygy}) &= 10^{-10} \\ &\vdots \end{aligned}$$

Joint and marginal probabilities. Things get more interesting when we deal with more than one random variable. For example, suppose our experiments are words spoken during a debate, and W is again the words spoken, while $S \in \{\text{Biden}, \text{Trump}, \ldots\}$ is the person speaking. We can talk about the *joint distribution* of S and W, written $P(S, W)$, which should satisfy

$$\sum_{s,w} P(S = s, W = w) = 1.$$

Let's make up some numbers:

$$\begin{aligned} P(S = \text{Trump}, W = \text{bigly}) &= 0.2 \\ P(S = \text{Trump}, W = \text{huge}) &= 0.4 \\ P(S = \text{Biden}, W = \text{c'mon}) &= 0.3 \\ P(S = \text{Biden}, W = \text{man}) &= 0.1. \end{aligned}$$

We also have to have

$$P(S = s) = \sum_w P(S = s, W = w)$$
$$P(W = w) = \sum_s P(S = s, W = w),$$

known as marginal distributions.

Using our made-up numbers, we have

$$P(S = \text{Trump}) = 0.2 + 0.4 = 0.6$$
$$P(S = \text{Biden}) = 0.3 + 0.1 = 0.4$$

and

$$P(W = \text{bigly}) = 0.2$$
$$P(W = \text{huge}) = 0.4$$
$$P(W = \text{c'mon}) = 0.3$$
$$P(W = \text{man}) = 0.1.$$

It's extremely common to write $P(w)$ as shorthand for $P(W = w)$. This leads to some sloppiness, because the symbol P is now "overloaded" to mean several things and you're supposed to know which one. To be precise, we should distinguish the distributions (using $P(S = s)$ or $P_S(s)$). But in NLP, we deal with some fairly complicated structures, and it becomes messy to keep this up. In practice, it's rarely a problem to use the sloppier notation.

Conditional probabilities. We also define the *conditional distributions*

$$P(s \mid w) = \frac{P(s, w)}{P(w)}$$
$$P(w \mid s) = \frac{P(s, w)}{P(s)}.$$

Note that

$$\sum_s P(s \mid w) = 1$$
$$\sum_w P(w \mid s) = 1.$$

You should know this already, but it should be second nature, and in particular, be sure never to get $P(s \mid w)$ and $P(w \mid s)$ confused! Using our made-up numbers:

$$P(\text{Trump} \mid \text{bigly}) = 0.2/0.2 = 1$$
$$P(\text{bigly} \mid \text{Trump}) = 0.2/0.6 \approx 0.333.$$

Expected values. Finally, if a random variable has numeric values, we can talk about its average or expected value. For example, let $c_{\mathrm{e}}(w)$ be the number of occurrences of the letter e in w. The *expectation* of c_{e} is

$$E[c_{\mathrm{e}}] = \sum_{w} P(W = w)\, c_{\mathrm{e}}(w),$$

and using our made-up numbers, this is

$$E[c_{\mathrm{e}}] = 0.2 \cdot 0 + 0.4 \cdot 1 + 0.3 \cdot 2 + 0.1 \cdot 0 = 1.$$

Estimating probabilities. There's a "true" probability distribution over English words, $P(W)$, but it's impossible to know what it really is. If we want actual numbers, we need an estimate: $P(w) \approx \theta_w$. (Here we write $P(w)$ for the true probability and θ_w for its estimate, but when we don't need to be so careful, we often just write $P(w)$ for the estimate.) We can obtain an estimate from a collection of English text, $w_1 \cdots w_N$. Let $c(w)$ be the number of times that word w is seen in the data. Then the *maximum-likelihood estimate* for $P(w)$ is:

$$\theta_w = \frac{c(w)}{\sum_{w'} c(w')} = \frac{c(w)}{N}.$$

It's called the maximum-likelihood estimate because it's the estimate that maximizes the *likelihood*,

$$\mathcal{L}(\Theta) = \theta_{w_1} \cdots \theta_{w_N}.$$

The likelihood is just the probability (estimate) of the data, but thought of as a function of Θ, which is the set of all the θ_w's. Maximizing it gives us the model that gives the most probability to the observed data.

1.4.2 Logarithms

You learned logarithms a long time ago, but you'll really use them a lot in this class. The following identities should be second nature:

$$\begin{aligned}
\log \exp x &= x & \exp \log x &= x \\
\log xy &= \log x + \log y & \exp(x + y) &= \exp x \exp y \\
\log \prod_i x_i &= \sum_i \log x_i & \exp \sum_i x_i &= \prod_i \exp x_i \\
\log x^n &= n \log x & \exp nx &= (\exp x)^n \\
\log 1 &= 0 & \exp 0 &= 1
\end{aligned}$$

Unless otherwise indicated, log and exp will always have base e.

Log-probabilities. Logarithms are used a lot to simplify expressions like this product of many probabilities:

$$p(x_1, \ldots, x_n) = \prod_i p(x_i).$$

It's extremely common to take the log of everything, changing the product into a sum:

$$\log p(x_1, \ldots, x_n) = \sum_i \log p(x_i).$$

There are a few of reasons for this. First, it used to be that additions are faster than multiplications, but we don't worry about this anymore (in fact, floating-point multiplication is sometimes faster). Second, it's often easier on paper to work with sums instead of products. (For example, taking derivatives is easier.)

Third, a product of many probabilities quickly becomes a very small number. An IEEE 754 double only goes down to 10^{-308}, and we often deal with probabilities much smaller than that. To avoid underflow, the typical solution is to use log-probabilities.

Computing with log-probabilities is easy. If we have two log-probabilities $\log p$ and $\log q$, instead of multiplying p and q, we add $\log p$ and $\log q$ (because $\log pq = \log p + \log q$). To compare p and q, just compare $\log p$ and $\log q$, which is equivalent.

The only tricky part is addition. To compute $\log(p + q)$ given $\log p$ and $\log q$, we can't do this:

$$\log(p + q) = \log(\exp \log p + \exp \log q)$$

because either of the exp's might cause an underflow. What should you do instead? The short answer is that you should use library functions designed for this purpose (in PyTorch, `torch.logaddexp` or `torch.logsumexp`).

The long answer is: Assume that $p > q$; if not, swap them. Then, observe that:

$$\begin{aligned}
\log(p + q) &= \log p \left(1 + \frac{q}{p}\right) \\
&= \log p + \log \left(1 + \frac{q}{p}\right) \\
&= \log p + \log \left(1 + \exp \log \frac{q}{p}\right) \\
&= \log p + \log(1 + \exp(\log q - \log p)).
\end{aligned}$$

Now, the exp could still cause an underflow, but the underflow is harmless. (Why?) For an extra little boost in accuracy, you can use the `log1p` function, found in nearly all standard libraries, which computes $\log(1 + x)$ but is accurate for small x. This is sometimes called the *log-sum-exp trick*.

Note that if p is a probability, $\log p$ is negative or zero. Sometimes we work with $-\log p$, which is positive or zero, but is confusingly called a *negative log-probability*.

Softmax. Suppose that $x_1, x_2, \ldots, x_n$ are log-probabilities; then their exps should sum to one. But sometimes, it's more convenient to let them be unconstrained numbers (called *logits*) and force them to sum to one by dividing by their sum. In neural networks, this is called a *softmax*. Let $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^\top$ be a vector of real numbers (positive or negative), and define softmax $\mathbf{x}$ to be the vector

$$[\operatorname{softmax} \mathbf{x}]_i = \frac{\exp x_i}{\sum_{i'=1}^n \exp x_{i'}},$$

where $\exp \mathbf{x}$ means the elementwise exp of $\mathbf{x}$. It's also common to express this as

$$\log\,[\operatorname{softmax} \mathbf{x}]_i = x_i - \log \sum_{i'=1}^{n} \exp x_{i'},$$

where the second term is computed using the trick above.

Bibliography

Bertsekas, Dimitri P. and John N. Tsitsiklis (2008). *Introduction to Probability*. 2nd. Athena Scientific.