Deep Learning for Natural Language Processing

IRS ML Section*

Hwa Chong Institution

1 Introduction to Natural Language Processing

1.1 What is NLP?

In its most general form, NLP is the study of allowing machines to understand language as it is used by humans. Note that this document will largely concern the use of deep learning to extract the meaning of text, which is the focus of most current NLP research.

1.2 Why do we need NLP?

Natural language analysis. Much of human-generated data is in the form of text - after all, tens of billions of text messages are sent every day, in addition to the myriad text-based social media posts made by hundreds of millions of users worldwide. As with other fields of data analysis, when data is generated, people would like to draw insights from it. Since the sheer volume of textual data precludes manual processing, data scientists have naturally turned to computers to analyse it.

Natural language generation. Several language-related tasks involve not just analysing human language but also generating it. The latter tends to be even more time-consuming for humans, hence the desire to automate it as well. As it turns out, natural language generation is typically also more difficult than pure language analysis from a computational perspective.

Human-computer interaction. By endowing computers with the ability to understand (or at least process) human language, we can and have significantly streamlined the way we use computers. Thanks to applications of NLP in human-computer interaction, most owners of modern smartphones now possess convenient alternatives to button-pushing, in the form of Siri or Google Assistant.

1.3 Some applications of NLP

- Sentiment analysis
- Speech recognition

* Written by Ho Wing Yip and Marcus Wee

- Fake news detection
- Machine translation
- Text summarization
- Question answering

2 Data Preprocessing

Before applying machine learning to language data - or for that matter, any kind of data - it often helps to process the data beforehand. Preprocessing language data usually serves to reduce it down to its most salient features, decreasing the complexity of the problem we are trying to solve. In this section, we detail a number of techniques for doing so.

2.1 Punctuation removal

To allow our models to analyse our data as easily as possible, we aim to distil the language down to its bare meaning, removing all extraneous noise. Consider the following un-punctuated sentence:

I prefer not to eat meat it just doesn't taste fantastic and environmentalists say it's bad for the planet anyway

While it obviously doesn't read very elegantly, one can easily discern what the person is trying to say. To machine learning models that aim to infer the meaning of a piece of text, punctuation is thus little more than a distraction most of the time. Hence, punctuation is typically removed during preprocessing.

2.2 Lowercase conversion

In a similar vein, whether a word is in uppercase or lowercase is often unimportant in NLP, though this varies between tasks. As such, we convert text to lowercase where applicable.

2.3 Tokenization

For humans, an intuitive starting point for analysing text is to split it into its constituent sentences, phrases, words and perhaps even characters. The same principle is applied in NLP, where processing a piece of text begins by splitting it into its linguistic building blocks, or *tokens*. This splitting process is termed *tokenization*.

Typically, language models operate on more granular tokens such as words, subwords or characters. Such tokens tend to recur more often than "coarser" tokens like sentences, making for more consistent and widely applicable analysis.

Word-level tokenization:
`Mary / had / a / beautiful / little / lamb / with / sparkling / eyes`

Subword-level tokenization:
`Mary / had / a / beauti / ful / little / lamb / with / sparkl / ing / eyes`

Character-level tokenization:
`M / a / r / y / h / a / d / a / b / e / a / u / t / ...`

Fig. 1: Various types of tokenization

2.4 Stopword removal

In a similar vein to punctuation removal, one often finds that certain words are of little use in determining the meaning of a piece of text. Consider the following sentence:

I prefer not to eat meat - it just doesn't taste fantastic, and environmentalists say it's bad for the planet anyway.

Now, let us remove some words:

I prefer not eat meat doesn't taste fantastic environmentalists say bad for planet

Despite the lack of some prepositions like 'to' and pronouns like 'it', the meaning of the text remains relatively clear - can you still infer the two reasons why the speaker does not wish to eat meat?

Words like the ones removed above are known as *stopwords*. They are commonly omitted during preprocessing as they are typically unimportant in determining the meaning of the text, as demonstrated above. Note that stopwords may vary between tasks - for instance, when determining the topic of a piece of text, it is usually safe to ignore the word 'not', as the subject matter being discussed is unlikely to depend on it. In contrast, if one wishes to examine the sentiment of the text, omitting 'not' can cause one's prediction to be the complete opposite of the actual opinion expressed. It is thus important to be discerning when applying NLP preprocessing techniques, for there is no one-size-fits-all preprocessing solution for all tasks.

Most NLP toolkits like NLTK and spaCy implement a list of common stopwords for several languages. You may also choose to implement your own stopword list if you find it comprehensive enough.

2.5 Stemming and lemmatization

Stemming and lemmatization are two ways to reduce various inflections (e.g. 'walks', 'walking', 'walked') of the same root word to a common form (e.g. 'walk'). This can reduce the size of our vocabulary - and thus the complexity of our task

- without sacrificing semantic information, as the meaning of words typically remains the same no matter which grammatical form they are in.

Stemming is the process of removing or replacing common word affixes; in English, examples include '-ing', '-ed', and '-ful'. The stemmed word need not be a valid word in the language. It is important to note that stemming is performed only on **single words** with **no regard for context**, making it a rather naive approach. The most popular stemming algorithm is Porter's algorithm, a full description of which can be found at http://snowball.tartarus.org/algorithms/porter/stemmer.html.

While stemming is usually an acceptable rudimentary solution, such an approach is too simplistic to deal with the fickle variety present in languages like English. As an example, consider the fact that the root form of 'better' is not 'bett' but 'good'. Thus, when possible, we turn to a more complete method - lemmatization.

Lemmatization converts words to their root form or *lemma*, taking into account **context** and **meaning**. Consider the word 'meeting' - depending on context, it could be either a noun in its root form, or the '-ing' form of the verb 'meet'. An ideal lemmatizer uses context to select the appropriate lemma. Many popular NLP packages like NLTK and spaCy have lemmatizers built in.

TODO: Provide sample code for preprocessing libraries (nltk, Spacy)

3 NLP Models

3.1 One-hot encoding

Now that we've cleaned our data, we need to make it machine-readable. Each word, or *token*, must thus be converted into a numerical form. Since the set of possible words is (at present) discrete, we represent each word by a one-hot-encoded vector. Denoting the set of all distinct words in our data (i.e. our *vocabulary*) by V, word i is represented by a $|V|$-element vector containing a 1 in position i and zeroes everywhere else.

Example. Suppose our vocabulary consists of the five words 'apple', 'banana', 'i', 'like' and 'no'. We could convert them to one-hot vectors as follows:

Each document then becomes a sequence of vectors. This can be represented by a matrix consisting of the one-hot representations of all words in the document simply stacked on top of each other. Fig. 3 illustrates this for the sentence "i no like banana".

Note: It is also useful to have an *out-of-vocabulary (OOV) token* to represent words not in the vocabulary. Such words may be encountered in test data, and in real-world model use.[1]

[1] In TensorFlow, OOV tokens are typically represented as a word with index 0. The vocabulary size increases by one, and one-hot encoding proceeds as usual, except

Index	Word	One-hot representation
1	apple	$[1, 0, 0, 0, 0]$
2	banana	$[0, 1, 0, 0, 0]$
3	i	$[0, 0, 1, 0, 0]$
4	like	$[0, 0, 0, 1, 0]$
5	no	$[0, 0, 0, 0, 1]$

Fig. 2: A possible one-hot representation of a 5-word vocabulary

$$\begin{matrix} \text{i} \\ \text{no} \\ \text{like} \\ \text{banana} \end{matrix} \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 3: The matrix representation of "i no like banana". Each row is the one-hot representation of its associated word.

3.2 Word embedding

Since we represent words as vectors, it makes sense to use the dot product as a way to measure similarity between words. The intuition behind this is as follows: if two word vectors are "similar" in the sense that they point in the same direction, their dot product will be positive. If the two words are "unrelated" in that their vectors are orthogonal, the dot product will be zero. If the word vectors point in opposite directions, making them "opposites", their dot product will be negative.

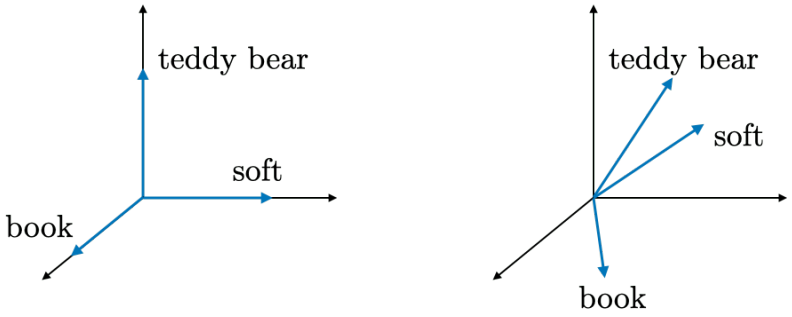
From this viewpoint, however, our current one-hot representation is still rather naive, in that it encodes no information about word similarity. If we use the dot product as a word similarity measure, all our words would be considered unrelated to each other, which is obviously not accurate! As such, NLP models typically use a process called *embedding* to generate word representations that are semantically richer than one-hot vectors.

We embed the one-hot vectors by linearly projecting them to an *embedding space*, where similar words should correspond to vectors pointing in similar directions. The linear projection is performed by multiplying the one-hot vectors by an *embedding matrix*, denoted by E. The dimension of the embedding space is a hyperparameter, which we shall call e.[2] Since our vocabulary has size $|V|$, each one-hot vector will have $|V|$ elements. Hence, E will be an $e \times |V|$ matrix.

with the first word having a 1 in position 2, the second word having a 1 in position 3, and so on. Note that the programmatic *index* of the 1 remains the same, as indexing in Python is zero-based.

[2] Typically, e is on the order of 10^1 to 10^2.

It is important to note that the model **learns** the embedding matrix, so as to generate the best possible word representations for the task at hand. There also exist pretrained embedding weights learned for a single task that have proven effective for several other applications.



One-hot word representation Embedded word representation

Fig. 4: Illustration of one-hot vs. embedded word representations. Image credit: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Learning embeddings. Normal layer weight learning through backpropagation, word2vec, skip-gram, negative sampling, GloVe, etc.

3.3 Cosine Similarity

As mentioned before, vector representation of words would allow us to make similarity judgements using the dot product. This is usually done in the form of cosine similarity which takes the dot product of the unit vectors to obtain the cosine of the angle between the two vectors. If the cosine of the angle is close to 0, then the words have are completely unrelated. If the cosine of the angle is close to 1, then the words are very similar and if the cosine of the angle is close to -1 then they have a completely opposite meaning.

However the effectiveness of using cosine similarity is heavily dependent on the effectiveness of vectorisation. Hence the choice of vectorization of words is extremely crucial.

3.4 Model architectures for NLP

RNNs This segment is trivial and left as a research exercise to the reader.

Transformers This segment is trivial and left as a research exercise to the reader.

CNNs This segment is trivial and left as a research exercise to the reader. Please do not use CNNs for this.

- High-dimensional embedding
- Model architectures, e.g. RNN, transformer, convolutional (less popular)
- Using pretrained models (e.g. Huggingface transformers) - typically you won't have enough computing power to build a SOTA model of your own