

Natural Language Processing

Muchang Bahng

August 29, 2024

Contents

We will use NLTK, PyTorch. Natural language processing refers to the use of machine learning to learn language tasks. The problems are very broad and wide, including:

1. predictive typing
2. speech recognition
3. handwriting recognition
4. spelling/grammar correction
5. authorship identification
6. machine translation
7. summarization
8. dialogue
9. etc.

Obviously these are not simply just the “ChatGPT”-like tasks that we are familiar with today, but use ideas from computer vision and other fields to develop better models. At the basis of a lot of these problems is **language modelling**, which will be the bulk of these notes.

1 Basics

1.1 Regular Expressions

Regular expressions allow us to find patterns in strings. In Python, the **r** in front of the string stands for **raw strings**, which literally takes in special characters (e.g. escape characters). Below, we search for the string **and**.

```
import re

data = """Natural Language Processing (NLP) is an interdisciplinary field that empowers
67 machines to understand, interpret, and generate human language. Its 4 applications
span across various domains, including chatbots, language translation, sentiment
analysis, and information extraction. We're going to rock'n'roll in the long-term. """

pattern = re.compile(r"and")
matches = pattern.finditer(data)

for match in matches:
    print(match)

# <re.Match object; span=(100, 103), match='and'>
# <re.Match object; span=(116, 119), match='and'>
# <re.Match object; span=(255, 258), match='and'>

print(data[100:103], data[116:119], data[255:258])
# and and and
```

There are special characters that allows us to identify other more specific patterns. We list them below.

- | | |
|----|---------------------------------|
| . | - Any Character Except New Line |
| \d | - Digit (0-9) |
| \D | - Not a Digit (0-9) |

<code>\w</code>	- Word Character (a-z, A-Z, 0-9, _)
<code>\W</code>	- Not a Word Character
<code>\s</code>	- Whitespace (space, tab, newline)
<code>\S</code>	- Not Whitespace (space, tab, newline)
<code>\b</code>	- Word Boundary
<code>\B</code>	- Not a Word Boundary
<code>^</code>	- Beginning of a String
<code>\$</code>	- End of a String
<code>[]</code>	- Matches Characters in brackets
<code>[^]</code>	- Matches Characters NOT in brackets
<code> </code>	- Either Or
<code>()</code>	- Group

1.2 Preprocessing

1.2.1 Tokenization and Stop Words

Tokenization is the process of dividing up a corpus or a complex sentence into words, also known as tokens. Using a streamlined series of regular expression evaluation, we are able to detect various words to tokenize. Note the following facts:

1. Uppercase and lowercase versions of the same word (e.g. `Language` vs `language`) are two different tokens.
2. Punctuations and special characters also count as a token (e.g. `.`, `,`, `(`, `)`).
3. Hyphenated words and some words with apostrophes are not separated, even though both components are valid words, since they are meant to be used together (e.g. `rock'n'roll` or `long-term`).
4. Some words with apostrophes that are abbreviations are indeed separated (e.g. `We're` to `We`, `'re`).

```

from nltk.tokenize import sent_tokenize, word_tokenize

example_string = """Natural Language Processing (NLP) is an interdisciplinary
field that empowers machines to understand, interpret, and generate human
language. Its applications span across various domains, including chatbots,
language translation, sentiment analysis, and information extraction. We're
going to rock'n'roll in the long-term. """

print(sent_tokenize(example_string))
# ['Natural Language Processing (NLP) is an interdisciplinary field that empowers
# machines to understand, interpret, and generate human language.', 'Its
# applications span across various domains, including chatbots, language
# translation, sentiment analysis, and information extraction.', "We're going to
# rock'n'roll in the long-term."]

print(word_tokenize(example_string))
# ['Natural', 'Language', 'Processing', '(', 'NLP', ')', 'is', 'an',
# 'interdisciplinary', 'field', 'that', 'empowers', 'machines', 'to',
# 'understand', ',', 'interpret', ',', 'and', 'generate', 'human', 'language',
# '.', 'Its', 'applications', 'span', 'across', 'various', 'domains', ',',
# 'including', 'chatbots', ',', 'language', 'translation', ',', 'sentiment',
# 'analysis', ',', 'and', 'information', 'extraction', '.', 'We', "'re",
# 'going', 'to', "rock'n'roll", 'in', 'the', 'long-term', '.']

# 0.004939079284667969 seconds

```

Sometimes, there are words that are used so often that it is pointless to have them (e.g. **the**, **an**, **a**, etc.). We don't want these words to take up space in our database, so we can use NLTK to store a list of words that we consider to be stop words. The default library of **stop words** in multiple languages can be downloaded with the following command, and we can execute them on the paragraph above. In NLTK 3.8.1, there are a total of 179 stop words in English, and you can add or remove the words in this file manually.

```

nltk.download("stopwords")
from nltk.corpus import stopwords

print(stopwords.words("english"))

# ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're",
# "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he',
# 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's",
# ...
# 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn',
# "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn',
# "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]

```

To filter the stopwords, we can loop over the words in the token list and remove it if it is a stop word.

1.2.2 Lemmatization and Stemming

Stemming is the process of producing morphological variants of a root word. For example, a stemming algorithm reduces the words “chocolates”, “chocolatey”, and “choco” to the root word “chocolate.” It helps to reduce different variants into a unified word for easier retrieval of data.

```

from nltk.stem import PorterStemmer

ps = PorterStemmer()
words = ["program", "programs", "programmer", "programming", "programmers"]
for w in words:
    print(w, " : ", ps.stem(w))

# program : program
# programs : program
# programmer : program
# programming : program
# programmers : program

```

1.3 Syntactic Structure

1.3.1 Parts of Speech

The parts of speech can be defined with varying degrees of refinement. First, let us get the basics down. Parts of speech fall into two broad categories: **closed class** and **open class**. Closed classes are those with relatively fixed membership, such as prepositions (new prepositions are rarely coined). However, nouns and verbs are open class since new nouns and verbs (e.g. iPhone or to fax) are always being created.

Class	Tag	Description	Example
Open Class	ADJ	Adjective: noun modifiers describing properties	<i>red, young, awesome</i>
	ADV	Adverb: verb modifiers of time, place, manner	<i>very, slowly, home, yesterday</i>
	NOUN	Words for persons, places, things, etc.	<i>algorithm, cat, mango, beauty</i>
	VERB	Words for actions and processes	<i>draw, provide, go</i>
	PROPN	Proper noun: name of a person, organization, place, etc.	<i>Regina, IBM, Colorado</i>
	INTJ	Interjection: exclamation, greeting, yes/no response, etc.	<i>oh, um, yes, hello</i>
Closed Class	ADP	Adposition (Preposition/Postposition): marks a noun's spatial, temporal, or other relation	<i>in, on, by, under</i>
	AUX	Auxiliary: helping verb marking tense, aspect, mood, etc.	<i>can, may, should, are</i>
	CCONJ	Coordinating Conjunction: joins two phrases/clauses	<i>and, or, but</i>
	DET	Determiner: marks noun phrase properties	<i>a, an, the, this</i>
	NUM	Numeral	<i>one, two, first, second</i>
	PART	Particle: a function word that must be associated with another word	<i>'s, not, (infinitive) to</i>
	PRON	Pronoun: a shorthand for referring to an entity or event	<i>she, who, I, others</i>
	SCONJ	Subordinating Conjunction: joins a main clause with a subordinate clause such as a sentential complement	<i>that, which</i>
Other	PUNCT	Punctuation	<i>;, ()</i>
	SYM	Symbols like \$ or emoji	<i>%, %</i>
	X	Other	<i>asdf, qwfg</i>

Table 1: Class, Tag, Description, and Example

Now there are two prominent frameworks used in the field of syntax to describe the structure of sentences in natural languages: **Phrase Structure Grammar (PSG)** and **Dependency Grammar (DG)**. They differ in their underlying principles, so we will introduce them separately.

1.3.2 Phrase Structure/Context Free Grammar

We have our starting units as words

the	cat	cuddly	by	door
DET	NOUN	ADJ	PREP	NOUN

Now these words can combine into phrases. A **noun phrase** (NP) refers to a phrase describing a noun; a **verb phrase** (VP) refers to a phrase describing a verb, and a **prepositional phrase** (PP) refers to describing a noun's spatial, temporal, or other relation.

the cuddly cat	by [the door]	walking over there
NP	PP NP	VP

From the example above, we can see that these phrases can be nested within each other. For example,

the	cuddly	cat	by	the	door
[NP]	[NP]
			[PP]
[NP]

We can make rules for how NP, PP and VPs are structured. By looking at **the cuddly cat**, we can make the following rule

NP --> DET ADJ NOUN

The adjective can be there or not, so in linguistics we have the notation (ADJ)*. Furthermore, if we attach a prepositional phrase at the end, we can have some structure like

NP --> DET (ADJ)* NOUN (PP)

which is a generalization of the first rule. A prepositional phrase such as **by the door** has the structure

PP --> PREP DET NOUN

A verb phrase like **talked to the cat** has the structure

VP --> VERB PREP DET NOUN

These set of rules is called the **grammar**, and they determine what phrases we are or aren't allowed to use. If we have another phrase like **the cat walked behind the dog**, which is a valid sentence, we need to account for this validity by adding the extra grammar rules

the	cat	walked	behind	the	dog
[NP]	[VP]
[NP]	[VERB]	[PP]
DET	NOUN	VERB	PREP	DET	NOUN

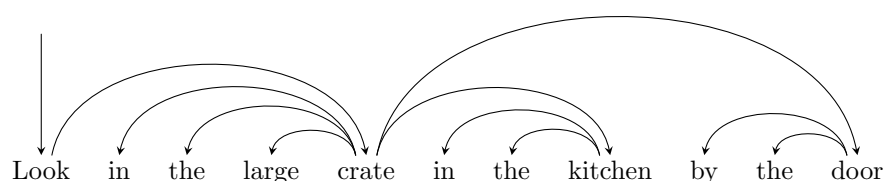
But there are a countless variety of sentences out there, and we can't keep making rules forever.

1.3.3 Dependency Grammar

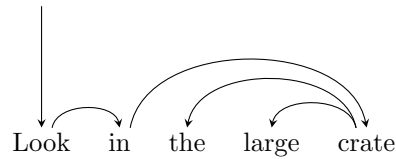
While phrase structure grammar is a popular paradigm for syntax construction, starting from the early 2000s, NLP researchers have swung behind dependency grammar. Given a phrase or a sentence, the **dependency structure** shows which words depend on (modify, attach to, or are arguments of) which other words. Humans communicate complex ideas by composing words together into bigger units to convey complex meanings. Listeners need to work out what modifies what, and similarly, a model needs to understand sentence structure in order to be able to interpret language correctly.

Let us start off with understanding what it means for a word to be dependent on another. Informally, a word A is a dependent of word B if B is needed to complete the phrase of A.

The basic idea is given a sentence, we want to take each word and find out its part of speech and what other words modify it. This creates a **dependency tree**.



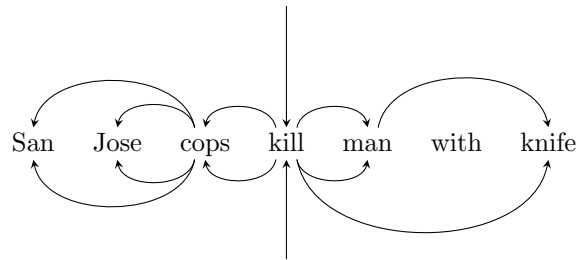
The dependency structure answers questions like “which crate?” or “where are you looking?” It may be more natural for us to have **in** as a dependency of **look**, with **crate** being the dependency of **of in**, as such



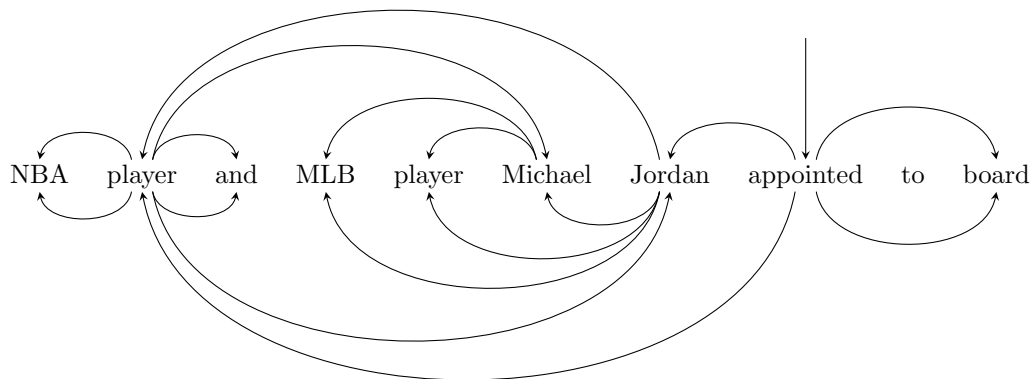
However, in the **universal dependency structure**, which aims to create these for many human languages, the design decisions led to the other convention.

Dependency structure is especially important to minimize ambiguity, as in the following phrases:

1. *Prepositional Phrase Attachment Ambiguity*: The sentence can have two meanings: The police in San Jose kill a man who has a knife, or the police in San Jose use a knife to kill a man.



2. *Coordination Scope Ambiguity*: We don't know whether Michael Jordan is both an NBA and MLB player, or if an NBA player is separate.



3. *Verb Phrase Attachment Ambiguity*: For example, “mutilated body washes up on Rio beach to be used for Olympics beach volleyball.”
4. *Adjective/Adverbial Modifier Ambiguity*

1.3.4 Dependency Conditioning Preferences

Now comparing these two paradigms, it may seem that writing a set of grammar rules may be more efficient than using a collection of dependency trees, called **tree banks**. After all, having a set of general grammar rules certainly seems easier than getting people to annotate a bunch of text with their parts of speech and modifiers. But treebanks are good since we can reuse the labor (many parsers and parts-of-speech taggers can be built on top of it), it gives broad coverage and not just a few intuitions, and it gives frequencies and distributional information of these words. Furthermore, there are other more useful sources of information:

1. *Bilxial affinities*: The dependency is plausible
2. *Dependency Distance*: Most dependencies are between nearby words.
3. *Intervening Material*: Dependencies rarely span intervening words or punctuation.
4. *Valency of Heads*: How many dependents on which side are usual for a head? e.g. given a noun, what kind of dependents would it have usually? It would normally have a determiner to the left (and almost never to the right) and an adjective to the left also. Perhaps a verb to the right.

Furthermore, there are some constraints on these dependencies, such as only one word is a dependent of ROOT and we don't want any cycles. This makes dependencies a tree, hence the name. The final issue is whether arrows can cross (be non-projective) or not. The definition of a **projective parse** is that there are no crossing dependency arcs when the arrows are laid out in their linear order, which all arcs above the words. Usually, we want the arrows to be projective so that we have a nested phrase structure like we saw in context free grammar, and most syntactic structure is projective, but dependency theory normally does allow non-projective structures to account for displaced constituents.

Now as for building dependency parsers, there are many ways to do this:

1. *Dynamic Programming* (Eisner 1996) gives a clever algorithm with complexity $O(n^3)$ by producing parse items with heads at the ends rather than in the middle.
2. *Graph Algorithms*: You create a minimum spanning tree for a sentence.
3. *Constraint Satisfaction*: Edges are eliminated that don't satisfy hard constraints.
4. *Transition-Based Parsing* or *Deterministic Dependency Parsing* uses a greedy choice of attachments guided by good machine learning classifiers.

2 Embeddings

Our goal is to create some embedding method that maps the vocabulary of words into some vector space \mathbf{V} . One could simply one-hot-encode all the words, but this is memory inefficient and the structure of the words are not captured. That is, we would like some associated metric d that tells us how similar two words are. At this point, it is not clear that this similarity entails, whether it'd be similar definitions (dog and canine), similar in groupings (e.g. dog and cat), or similar in context (doctor and scalpel).

There have been many attempts to create this mapping, but the most successful by far has followed the idea of **distributional semantics**. This hypothesis states that words that occur in similar contexts tend to have similar meanings.

2.1 Frequency Semantics

2.1.1 Term-Document Matrix

Let's start with the simplest distributional model based on the **co-occurrence matrix**, which represents how often words co-occur.

Definition 2.1 (Term-Document Matrix). Given $\mathcal{D} = \{D_1, \dots, D_m\}$ documents and $\mathcal{V} = \{v_1, \dots, v_n\}$ total words (tokens) in the document, the **term-document matrix** is a $n \times m$ matrix where the ij th entry represents the number of times token v_i occurred in document D_j . One can see how this will be a sparse matrix, since there may be many infrequent words that appear only once in exactly one document.

Example 2.1. Given the following three documents

D_1 : I like deep learning.

D_2 : I like NLP.

D_3 : I enjoy flying.

we can construct a term-document matrix as

	D_1	D_2	D_3
I	1	1	1
like	1	1	0
deep	1	0	0
learning	1	0	0
NLP	0	1	0
enjoy	0	0	1
flying	0	0	1

Table 2: Term-Document Matrix

Now there are two ways that we can analyze this matrix. First, note that the columns $M_{:,i}$ are vectors that represent the words in document D_i . This is known as the **bag-of-words model**. The rows $M_{j,:}$ represent the frequency of a certain word v_j in each document. Therefore, we can compare documents by comparing their corresponding vectors D_i, D_j which represent the distribution of its words, and we can compare words by comparing their corresponding vectors v_i, v_j , which represent their distribution over the documents.

Upon inspection you can notice that this is exactly feature extraction. Each document in the corpus is a data point and the count of specific words are the features. We can implement this with scikit learn's feature extractor, which gives the word count of a corpus, represented as a list of strings. We first load in our corpus.

```

from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd

corpus = [
    "I like deep learning.",
    "I like NLP.",
    "I enjoy flying."
]

cvectorizer = CountVectorizer()
X = cvectorizer.fit_transform(corpus)
terms = cvectorizer.get_feature_names_out()

# List of all terms in order of document-term matrix columns.
print(terms)
# ['deep' 'enjoy' 'flying' 'learning' 'like' 'nlp']

# Document term matrix
print(X.toarray())
# [[1 0 0 1 1 0]
#  [0 0 0 0 1 1]
#  [0 1 1 0 0 0]]

# Data type of matrix and shape (note this is a sparse matrix type)
print(type(X))      # <class 'scipy.sparse._csr.csr_matrix'>
print(X.shape)      # (3, 6)

documentTermMatrix = pd.DataFrame(X.toarray(),
                                   index=["Doc 1", "Doc 2", "Doc 3", ],
                                   columns=terms)

print(documentTermMatrix.to_string())

#           deep  enjoy  flying  learning  like  nlp
# Doc 1         1     0      0          1     1     0
# Doc 2         0     0      0          0     1     1
# Doc 3         0     1      1          0     0     0

```

2.1.2 Term-Term Matrix

Another way to compare words is through a term-term matrix.

Definition 2.2 (Term-Term Matrix). A **term-term matrix**, or a **co-occurrence matrix**, is a $|V| \times |V|$ matrix that represents the number of times the row (target, center) word and the column (context) word co-occur in some context in some training corpus.

1. The context could be the document, in which case the element M_{ij} represents the number of times the two words v_i, v_j appear in the same document.
2. It is most common to use smaller contexts, generally a window around the word, e.g. ± 4 words, in which case M_{ij} represents the number of times (in some training corpus) v_j appears around ± 4 words around v_i .

Example 2.2. Given a window length of 1 (most common for window length to be 5 through 10) and a

corpus of three documents:

D_1 : I like deep learning.

D_2 : I like NLP.

D_3 : I enjoy flying.

the co-occurrence matrix (note that it should be symmetric). is

	I	like	deep	learning	NLP	enjoy	flying
I	0	2	1	0	0	1	0
like	2	0	1	0	1	0	0
deep	1	1	0	1	0	0	0
learning	0	0	1	0	0	0	0
NLP	0	1	0	0	0	0	0
enjoy	1	0	0	0	0	0	1
flying	0	0	0	0	0	1	0

Table 3: Co-occurrence Matrix

This gives us a representation of words as **co-occurrence vectors**, which is similar to the word2vec embedding since they both use windows of a certain size.

2.2 Cosine Similarity

To measure similarity between two words or documents, it may be natural to take some sort of distance in $\mathbb{R}^{|V|}$. However, due to the size of the documents being nonuniform, we would like to measure how parallel one vector is to another. Two very similar documents D_i, D_j , where one is twice as long as the other, would be expected to have a proportional document vector satisfying $D_i \approx 2D_j$. Therefore, the angle between the two vectors would be best representation of the similarity metric.

$$\text{cosine}(v_i, v_j) = \frac{v_i \cdot v_j}{\|v_i\| \|v_j\|} = \cos \theta$$

2.3 TF-IDF and PMI

Let us focus on the vectors representing words, where the dimensions are documents. The words v_1, v_2 are essentially defined by their frequency in a corpus of documents D_1, \dots, D_m . The TF-IDF algorithm focuses on two main modifications of the frequency representation mentioned above.

1. *TF*: The raw frequency of the words may be a skewed representation of the token, since the difference between 0 and 1 occurrence in document D_i is not the same as the difference between 1000 and 1001 occurrences in another document. Unimportant stop words like “the” that occur frequently enough shouldn’t have as much of an effect on the representation of the word. We can filter these stop words out like before, but depending on context, there may be other words that are a less drastic impact. The additional impact of another instance of a word should have diminishing returns.
2. *IDF*: “Special” words that occur in only a few documents should have higher weights, since they are useful for discriminating those documents from the rest of the collection. Therefore, we would want this measure to be inversely proportional to the number of documents it is in. We can define

$$\frac{N}{\text{df}_t}$$

where N is the total number of documents and df_t is the number of documents that word t is in. Due to the large number of documents in a corpus, this value is also squashed down by a logarithm.

Given these two ideas, we define the two following forms of measure.

Definition 2.3 (TF). The **term frequency** is a measure of the frequency of the word t in the document d , squashed by the logarithm function (by diminishing returns) and adding 1 so that this weight is 0 when there are 0 occurrences.

$$\text{tf}_{t,d} = \log_{10} (\text{count}(t, d) + 1)$$

Definition 2.4 (IDF). The **inverse document frequency** is defined

$$\text{idf}_t = \log_{10} \left(\frac{N}{\text{df}_t} \right)$$

Definition 2.5 (tf-idf). The **tf-idf** weighted value for word t in document d thus combines the two values together.

$$w_{t,d} := \text{tf}_{t,d} \cdot \text{idf}_t$$

You can implement this in sklearn with the Tf-idf vectorizer, which again extracts features from a corpus. Let us use the same corpus as above. We can compare this matrix to the document-term matrix.

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

corpus = [
    "I like deep learning.",
    "I like NLP.",
    "I enjoy flying."
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)

tfidfMatrix = pd.DataFrame(X.toarray(),
                           index=["Doc 1", "Doc 2", "Doc 3", ],
                           columns=terms)

print(tfidfMatrix)
```

#	deep	enjoy	flying	learning	like	nlp
# Doc 1	0.622766	0.000000	0.000000	0.622766	0.473630	0.000000
# Doc 2	0.000000	0.000000	0.000000	0.000000	0.605349	0.795961
# Doc 3	0.000000	0.707107	0.707107	0.000000	0.000000	0.000000

An alternative weighting function to tf-idf is the PPMI (positive PMI), which is used for term-term matrices, when the vector dimensions correspond to words rather than documents. PPMI draws on the intuition that the best way to weigh the association between two words is to ask how much more the two words co-occur in our corpus than we would have a priori expected to appear by chance.

Definition 2.6 (PMI). The **pointwise mutual information** between a target word w and context word c is defined as

$$\text{PMI}(w, c) = \log_2 \frac{P(w, c)}{P(w) P(c)}$$

We can see that the numerator observes the joint distribution of the two words observed together, while the denominator represents their observation independently. We take the logarithm so that its range is $(-\infty, +\infty)$, where a value of greater than 0 indicates that the words co-occur more often and values less than 0 indicates that words are co-occurring less often than chance. However, negative PMI values tend to be unreliable since these probabilities can get very small, unless our corpus is enormous. Therefore, in practicality, we use the **positive PMI**, which just sets all negative values to 0.

$$\text{PPMI}(w, c) = \max \left\{ 0, \log_2 \frac{P(w, c)}{P(w) P(c)} \right\}$$

Given a term-term matrix, we can organize the PPMIs in a matrix. That is, let f_{ij} be the number of times word w_i occurs in context c_j .

2.4 Latent Semantic Analysis (LSA)

Clearly, we can see that text data suffers from many problems.

1. the vectors increase in size with vocabulary
2. it is very high dimensional and requires a lot of storage, though it is sparse. Even after preprocessing methods such as stop word removing, tokenization, and stemming, the document-term or TF-IDF matrices \mathbf{A} are extremely high dimensional.
3. subsequent classification models have sparsity issues, implying that models may be less robust

Our idea is to store the most important information in a fixed, small number of dimensions (ideally between 25 and 1000): a dense vector. A linear dimensionality reduction technique is to use principal component analysis. We basically take the SVD of the matrix, which reformulates the text data of r linearly uncorrelated **latent (hidden) features** through a **low-rank approximation**.

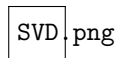
There are multiple motivations for this:

1. The original matrix may simply be too large to handle computationally and this approximation is a necessary evil.
2. The original matrix may be noisy, and anecdotal instances of terms are to be eliminated.
3. The original matrix may be overly sparse relative to the “true” matrix. That is, the original matrix lists only the words actually in each document, whereas we may be interested in all words *related* to each document.

We take the SVD of the matrix,

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

where \mathbf{U} and \mathbf{V} are unitary, with $\mathbf{\Sigma}$ diagonal. By rearranging the columns of \mathbf{U} and \mathbf{V} , we can have the diagonal entries (i.e. singular values) of $\mathbf{\Sigma}$ to go from greatest to least. Since the yellow part in the middle matrix is 0, the yellow of the right does not matter in the decomposition. Now to perform dimensionality reduction, we can remove the blue parts.



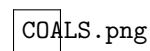
That is, we

1. Keep the top k right singular vectors $\mathbf{V} \mapsto \mathbf{V}_k$
2. Keep the top k left singular vectors $\mathbf{U} \mapsto \mathbf{U}_k$
3. Keep only the top k singular values $\mathbf{\Sigma} \mapsto \mathbf{\Sigma}_k$

The rank- k approximation is

$$\mathbf{A}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$$

After this, the documents and the words are compared by cosine similarity. It turns out that the normal error assumptions for SVD does not align with the structure of the raw counts data, which may be skewed with extremely frequent words (e.g. “the”). Therefore, we can focus on the TF-IDF matrix, which logs the frequencies, bounds some counts, or may ignore stop words. These kinds of models were explored a lot in the 90s or the 00s, with creative ideas like ramped windows that weighted closer words more than further ones, or used Pearson correlations instead of counts. In fact, Rohde’s COALS model showed interesting semantic properties between words that captured nice patterns between an action and an individual doing the action.



So by taking the vector differences we can say things like “drive is to driver as swim is to swimmer.”

2.5 GenSim Package (Generate Similar)

The Python package `gensim` is used to implement the word2vec algorithm. The core concepts of the package are:

1. *Document*: Some text represented by a string.
2. *Corpus*: A collection of documents.
3. *Vector*: A mathematically convenient representation of a document.
4. *Model*: An algorithm for transforming vectors from one representation to another.

2.6 Word2Vec

Note that the word representations are sparse. There are many computational tricks to work with sparse vectors, but there tends to be lots of noise and problems with them. Rather, we look at a more powerful word representation called **embeddings**, which are short dense vectors ranging in dimension from 50 to 1000. It turns out that dense vectors work better in every NLP task than sparse vectors.

The intuition for word2vec is that instead of counting how often each word w occurs near, say *apricot*, we'll instead train a classifier on a binary prediction task: "Is word w likely to show up near *apricot*?" We don't actually care about this prediction task. Instead, we will take the learned classifier weights as the word embeddings. Furthermore, we can just use running text as implicitly supervised training data. A context word c that occurs near *apricot* acts as a sample of "yes" to the question above.

Imagine a sentence like the following, with **target word** *apricot*, with a windows of ± 2 **context words**.

... lemon, a [tablespoon of apricot jam, a] pinch ...

Now our goal is to train a classifier where given the pair $\mathbf{x} = (\mathbf{w}, \mathbf{c})$, our output must be $y \in \{0, 1\}$, where 0 means not existing and 1 means within. This means that $y \mid \mathbf{x} \sim \text{Bernoulli}(\mu)$, where μ is dependent on (\mathbf{w}, \mathbf{c}) . This probability should be higher the more similar \mathbf{w} and \mathbf{c} are. We have already established that the similarity metric can be effectively represented by the dot product, we have

$$\mu = \mu(\mathbf{w}, \mathbf{c}) = \frac{1}{1 + \exp(-\mathbf{w} \cdot \mathbf{c})}$$

and so, the idea of our algorithm is that every time we find a window of context words around a target word, we should update both \mathbf{w} and \mathbf{c} s.t. they are more parallel.

2.6.1 Skip Gram with Negative Sampling

Let us formalize this concept a bit. Let us one-hot encode every word in our corpus to the set

$$\mathcal{V} = \{\mathbf{v}_1, \dots, \mathbf{v}_{|\mathcal{V}|}\}$$

Now, what we wish to do is to embed all these vectors in some \mathbb{R}^E , where E is a hyperparameter that we select and preferably $E < |\mathcal{V}|$. The embeddings \mathbf{w}_i for every $\mathbf{v}_i = \mathbf{e}_i$ can be simply stored as the columns of a $E \times |\mathcal{V}|$ matrix.

$$\begin{pmatrix} | & \dots & | \\ \mathbf{w}_1 & \dots & \mathbf{w}_{|\mathcal{V}|} \\ | & \dots & | \end{pmatrix} \begin{pmatrix} | \\ \mathbf{e}_i \\ | \end{pmatrix}$$

Let us denote this matrix $\boldsymbol{\theta}_w$. Therefore, our model given the one-hot encoded target and context words \mathbf{w}, \mathbf{c} is

$$\mu_{\boldsymbol{\theta}_w}(\mathbf{w}, \mathbf{c}) = \frac{1}{1 + \exp[-(\boldsymbol{\theta}_w \mathbf{w}) \cdot (\boldsymbol{\theta}_w \mathbf{c})]}$$

Now it turns out that the computations become easier if we have two embedding matrices, $\boldsymbol{\theta}_w$ for the target word \mathbf{w} and $\boldsymbol{\theta}_c$ for the context word \mathbf{c} , so our model really is

$$\mu_{\boldsymbol{\theta}}(\mathbf{w}, \mathbf{c}) = \frac{1}{1 + \exp[-(\boldsymbol{\theta}_w \mathbf{w}) \cdot (\boldsymbol{\theta}_c \mathbf{c})]}$$

where $\theta = (\theta_w, \theta_c)$. We can imagine these two matrices sort of “stacked” on top of each other.

Embeddings/word2vec_params.png

Now given whatever sequence of $\pm c$ context words around a target, we will make the simplifying assumption that these words occur independently, and so the probability that a sequence of words $\mathbf{c}_{t-m:t+m}$ occur around \mathbf{w}_t with window size m is

$$\mu_{\theta}(\mathbf{w}, \mathbf{c}_{t-m:t+m}) = \prod_{j=-m}^m \mu_{\theta}(\mathbf{w}, \mathbf{c}_j)$$

Now that we have our probabilistic model, we show how to generate our relevant training data. Consider the phrase

... lemon, a [tablespoon of apricot jam, a] pinch ...

with **apricot** as our target word. Just from this, we can generate 4 positive samples

$$\begin{aligned} \mathbf{x}^{(1)}, y^{(1)} &= (\text{apricot}, \text{tablespoon}), 1 \\ \mathbf{x}^{(2)}, y^{(2)} &= (\text{apricot}, \text{of}), 1 \\ \mathbf{x}^{(3)}, y^{(3)} &= (\text{apricot}, \text{jam}), 1 \\ \mathbf{x}^{(4)}, y^{(4)} &= (\text{apricot}, \text{a}), 1 \end{aligned}$$

where the words above would be in their one-hot encoded form. For training a binary classifier we need to have negative samples, too. Skip gram with negative sampling in fact uses more negative samples than positive ones (with the ration between them set by a parameter R). So for every positive sample, we pick a random noise word from the entire vocabulary, constrained not to be the target word \mathbf{w} . While this model could work with $R = 1$, it tends to perform better and is more stable if we have higher numbers, such as $R = 10, 15$. For example, if $R = 2$, then we would have 8 negative samples as such:

$$\begin{aligned} \mathbf{x}^{(5)}, y^{(5)} &= (\text{apricot}, \text{aardvark}), 0 \\ \mathbf{x}^{(6)}, y^{(6)} &= (\text{apricot}, \text{my}), 0 \\ \mathbf{x}^{(7)}, y^{(7)} &= (\text{apricot}, \text{where}), 0 \\ \mathbf{x}^{(8)}, y^{(8)} &= (\text{apricot}, \text{coaxial}), 0 \\ \mathbf{x}^{(9)}, y^{(9)} &= (\text{apricot}, \text{seven}), 0 \\ \mathbf{x}^{(10)}, y^{(10)} &= (\text{apricot}, \text{forever}), 0 \\ \mathbf{x}^{(11)}, y^{(11)} &= (\text{apricot}, \text{dear}), 0 \\ \mathbf{x}^{(12)}, y^{(12)} &= (\text{apricot}, \text{if}), 0 \end{aligned}$$

But these negative samples are not chosen randomly or according to counts. Rather, we sample them according to their **weighted unigram frequency**, usually of form

$$p_{\alpha}(c) = \frac{\text{count}(c)^{\alpha}}{\sum_v \text{count}(v)^{\alpha}}$$

and setting $\alpha = 3/4$ gives the best results in practice, since it gives rare noise words slightly higher probability so most of the probability measure wouldn't be dominated by stop words like “the”. This is a probabilistic method, as there may be times when the negative sample actually occurs in the context, but 99.99% of the time it will be a true negative sample.

To train the model, we should compute the likelihood of the entire training data. Given that we have our training data of form $(\mathbf{x}^{(n)}, y^{(n)}) = (\mathbf{w}^{(n)}, \mathbf{c}^{(n)}, y^{(n)})$ for $n = 1, \dots, N$, our likelihood for the entire dataset is

$$L(\theta) = \prod_{n=1}^N \mu_{\theta}(\mathbf{w}^{(n)}, \mathbf{c}^{(n)})^{y^{(n)}} (1 - \mu_{\theta}(\mathbf{w}^{(n)}, \mathbf{c}^{(n)}))^{1-y^{(n)}}$$

where we can take the negative average log and simplify further, but this is the extent that we will go for now.

$$-\ell(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{n=1}^N y^{(n)} \log(\mu_{\boldsymbol{\theta}}(\mathbf{w}^{(n)}, \mathbf{c}^{(n)})) + (1 - y^{(n)}) \log(1 - \mu_{\boldsymbol{\theta}}(\mathbf{w}^{(n)}, \mathbf{c}^{(n)}))$$

Obviously we can use optimization techniques to minimize this w.r.t. $\boldsymbol{\theta}$.

2.6.2 Continuous Bag of Words (CBOW)

To implement doc2vec, we utilize nltk for preprocessing and gensim to implement word2vec.

```
import string, nltk
from nltk.corpus import brown
from gensim.models import Word2Vec

nltk.download("brown")

# Preprocessing data to lowercase all words and remove single punctuation words
document = brown.sents()

data = []
for sent in document:
    new_sent = []
    for word in sent:
        new_word = word.lower()
        if new_word[0] not in string.punctuation:
            new_sent.append(new_word)
    if len(new_sent) > 0:
        data.append(new_sent)
```

This data would be a list, with each element a list of tokens making up a sentence. For example,

```
# Total number of sentences/documents in corpus = 57158
print(len(data))

# Total number of words in first sentence/document = 22
print(len(data[0]))

# First list of tokens
print(data[0])
# ['the', 'fulton', 'county', 'grand', 'jury', 'said', 'friday', 'an', 'investigation',
'of', 'atlanta's', 'recent', 'primary', 'election', 'produced', 'no', 'evidence', 'that',
'any', 'irregularities', 'took', 'place']
```

Now we create the word2vec model and train it with the initializer containing the following parameters. Note that this is a stochastic process, and refer to the documentation to learn how we can make this deterministic.


```

model = Word2Vec(
    sentences = data,      # list of list of tokens
    min_count = 5,        # Ignores all words with total frequency lower than this
    vector_size = 50,     # output vector dimensions
    window = 10,          # window of context words
    epochs = 20,          # number of epochs trained over dataset
    workers = 1,          # number of worker threads for multiprocessing
    sg = 0,               # 0 for CBOW, 1 for skip-gram
    hs = 0,               # 1 for hierarchical softmax, 0 for negative sampling
    negative = 5,          # num of negative samples per positive sample in skip-gram
    alpha = 0.025,        # training step
    min_alpha = 0.0001    # linearly decreases to this training step value
)

```

Now after training, this `Word2Vec` object `model` essentially has a dictionary with keys consisting of words in the corpus and values to be the embedding vectors.

```

print(model.wv["love"])

# [ 1.0745513 -1.8171308 -2.4329011 -0.3691842 -0.95292336 -0.54824775
#   1.1184701 -1.2525641 -0.7875846 -3.7816436 -1.341159  2.6486464
#  -0.30800238 -2.7417247  0.17696398 -2.9048784  1.621813  0.49121374
#   0.4354661 -1.6528435 -2.4828649  0.4085583 -0.7043962  2.8490443
#  -0.98837584  1.6951126 -1.607722  1.3588951 -0.03844598 -0.4779845
#  -3.2942739  1.3696849  0.07875736  1.0799417 -1.6086684  0.6993245
#   1.5824703  1.5176587  1.626068  1.7591808 -1.3893017 -2.4028397
#  -0.36541265  0.71958435  2.0678997 -1.6587187  1.6821662 -3.3152702
#  -1.6718794 -1.6396806 ]

```

2.6.3 Intrinsic Word Vector Evaluation

The `word2vec` performs quite well in preserving **semantic accuracy** (i.e. the meanings of the word). That is, it is true that the embedding $i \mapsto w_i$ often satisfies

$$w_{queen} = w_{king} - w_{male} + w_{female}$$

To quantify this analogy of $a : b = c : ?$, we want to find the word that is closest to $x_b - x_a + x_c$ in terms of cosine similarity. That is, we must find

$$d = \arg \max_i \frac{(x_b - x_a + x_c) \cdot x_i}{\|x_b - x_a + x_c\|}$$

However, there is the possibility that this information is here, but it just may not be linear. Other very interesting relations, such as male/female, or even company/CEO, can be found.

GloVe_Visual.png

Furthermore, we can capture **syntactic accuracy** (i.e. the grammar), as shown below using superlatives.

syntactic.png

2.7 Doc2Vec

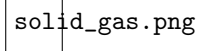
2.7.1 Soft Cosine Measure

2.7.2 Word Mover's Distance

2.8 Global Vectors (GloVe)

The initial goal of GloVe was to connect the linear algebra based embeddings like LSA and COALS with the iterative, neural-updating models like skip-gram and CBOW. It seemed like linear algebra methods were superior since they have fast training times and have efficient use of statistics, but they were primarily used to capture word similarity. However, the NN models scales with corpus size, generate improve performance on other tasks, and can capture complex patterns beyond word similarity, though they have an inefficient use of statistics.

It is nice to encode the **meaning components** between words. That is, the concept of vector additions and subtractions work as an analogy (e.g. male to female, king to queen or verb to agent such as truck to driver). A new insight is that the *ratios* of co-occurrence probabilities can also encode the meaning components. Say that you want to capture the meaning component from solid to gas. We can take the co-occurrence matrix of a document and compare the probabilities of certain words x appearing around the context of “ice” and “steam.” This gives us the matrix



We would like to capture the ratios of co-occurrence probabilities as linear meaning components in a word vector space. That is, the key property that we want is that the dot product of w_i and w_j represent the log probability of co-occurrence.

It turned out that a log-bilinear model fits this well, by setting

$$w_i \cdot w_j = \log \mathbb{P}(i | j)$$

with vector differences

$$w_x \cdot (w_a - w_b) = \log \frac{\mathbb{P}(x | a)}{\mathbb{P}(x | b)}$$

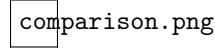
The GloVe model wanted to unify the co-occurrence model and the neural model, and so we wanted the dot product of two words i and j $w_i \cdot w_j$ to be similar to the log of the co-occurrence $\log X_{ij}$ (plus some bias), where X is the matrix. Therefore, the entire loss function would be

$$J = \sum_{i,j=1}^V f(X_{ij})(w_i \cdot w_j + b_i + b_j - \log X_{ij})^2$$

where we want our parameters $\theta = \{w_1, \dots, w_V\}$ to be adjusted so that $w_i \cdot w_j$ is as close as possible to the log of the co-occurrence matrix (with a bias term for both words). We also want to bound X_{ij} with f so that the loss isn't too dependent on a few very frequent words.

2.8.1 Performance Comparison

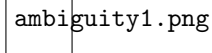
Let us compare the semantic and syntactic performance of the embedding algorithms we know so far. As we can see in the table below, unscaled, raw-count SVD does terribly, but with proper scaling (SVD-S) or with COALS (SVD-L), we can get decent scores even without a neural network. CBOW and Skip Gram (SG) performs even better, but GloVe is the best performer, with the ideal embedding dimension being around 300.



However, the author (and lecturer of the Stanford series I am working on) mentions that this outperformance may be due to having better data (Wikipedia is better than news text).

2.9 Word Ambiguity

It is often the case that one word may have multiple meanings. For example, the word “gay” may refer to homosexual or jolly. One simple solution is to train multiple embeddings for each definition of a word. The idea is that we cluster word windows around the words and retrain with each word assigned to multiple different clusters, e.g. $\text{bank}_1, \text{bank}_2$, etc.



This model works and aligns with our traditional sense of how these work, it tends to be imperfect since we’re trying to refine our sense of a word into k different parts. If we just keep the one-word one-embedding model, we find that different senses of the word reside in a “linear superposition” (weighted sum) of embeddings of its sub-meanings. For example,

$$\text{bank} = \alpha_1 w_{\text{bank}_1} + \alpha_2 w_{\text{bank}_2} + \dots + \alpha_k w_{\text{bank}_k}$$

where $\alpha_i = \frac{f_i}{f_1 + \dots + f_k}$ for frequency f . This may look useless at first glance since we just have the average meaning of these vectors, but because of ideas from **sparse coding**, you can actually separate out the senses! This is possible because in high-dimensional vector spaces, things tend to be extremely sparse.

3 Classical Learning Approaches

Some include sentiment analysis, and dependency structure.

3.1 Logistic Regression

3.2 Latent Dirichlet Allocation (LDA)

4 Sequence Tagging and Dependency Parsing

If we have a good word embedding model parts of speech or named entity recognition would be much easier. However, we will also regard it as a problem in its own right.

Dependency parsing is a natural language processing (NLP) technique that analyzes the grammatical structure of a sentence by determining the relationships between words. It aims to identify the syntactic dependencies among words and represent them as a directed graph, known as a dependency tree.

In dependency parsing, each word in a sentence is assigned a grammatical role (such as subject, object, modifier, etc.) and linked to its corresponding head word, which governs or controls it. The relationships between words are represented as labeled directed edges, where the head word is the parent node, and the dependent word is the child node.

For example, consider the sentence: “John eats an apple.” In dependency parsing, the word “eats” would be the head word, and “John” and “apple” would be its dependents. The word “eats” would have a dependency relation of “subject” with “John” and “object” with “apple.” This information can be represented as a dependency tree, where “eats” is the root, “John” is connected to “eats” with a subject edge, and “apple” is connected to “eats” with an object edge.

Dependency parsing has various applications in NLP, including syntactic analysis, information extraction, machine translation, sentiment analysis, and question answering. It helps in understanding the structure of a sentence, resolving ambiguities, and enabling more sophisticated language processing tasks. Dependency parsers can be trained using machine learning techniques, such as supervised learning or transition-based parsing algorithms, using annotated dependency treebanks.

Sentiment classification - Logistic, MLPs, RNNs (element-wise max or mean of all hidden states, Lec 6 34:17) Sequence Tagging - HMM, CRF, MLPs, RNNs Dependency Parsing - Language Encoder Module - Question Answering, Machine Translation Text Generation - i.e. RNN-LMs

5 Basic Language Models

Unlike classification problems, **language modeling** refers to the task of predicting upcoming words from prior word context. That is, given a string of words w_1, \dots, w_{t-1} , we want to provide a conditional probability distribution of

$$\mathbb{P}(w_t \mid w_{t-1}, \dots, w_1)$$

5.1 N Gram Model

An N-gram model basically models this as a giant Markov chain, which approximates the probability distribution of the next word as dependent on only the last $t - 1$ words.

$$\mathbb{P}(w_t \mid w_{t-1}, \dots, w_1) \approx \mathbb{P}(w_t \mid w_{t-1}, \dots, w_{t-N+1}) = \frac{\mathbb{P}(w_t, w_{t-1}, \dots, w_{t-N+1})}{\mathbb{P}(w_{t-1}, \dots, w_{t-N+1})}$$

In order to calculate this probability, we literally just count them in some large corpus of text. In order to construct such as model there are two functions that will be helpful: NLTK's `word_tokenize` and `ngrams` methods.

```
from nltk.tokenize import word_tokenize
from nltk import ngrams

sentence = "I love data science and machine learning."
tokens = word_tokenize(sentence)
# ['I', 'love', 'data', 'science', 'and', 'machine', 'learning', '.']

grams = list(ngrams(tokens, 3,
                    pad_left=True,
                    pad_right=True,
                    left_pad_symbol='<s>',
                    right_pad_symbol='</s>'
                    )
            )
# [('<s>', '<s>', 'I'), ('<s>', 'I', 'love'), ('I', 'love', 'data'), ('love', 'data',
# 'science'), ('data', 'science', 'and'), ('science', 'and', 'machine'), ('and',
# 'machine', 'learning'), ('machine', 'learning', '.'), ('learning', '.', '</s>'), ('.',
# '</s>', '</s>')]
```

Now this works for only one sentence, so if we take in a document with many sentences, we would like to construct a data structure that maps from the set of $N - 1$ previous words to the next word. From here we will create a very crude N-gram model.

```
import random, time
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk import ngrams

class NgramModel(object):

    def __init__(self, N:int, path:str):

        now = time.time()

        self.N = N
        self.context = {}

        with open(path, 'r') as f:
            sentences = sent_tokenize(f.read())

        for sentence in sentences:
            tokens = word_tokenize(sentence)

            grams = ngrams(tokens, N,
                           pad_left=True,
                           pad_right=True,
                           left_pad_symbol='<s>',
                           right_pad_symbol='</s>'
                           )

            for tup in grams:
                prev_words = tup[:-1]
                next_word = tup[-1]
                if prev_words in self.context:
                    self.context[prev_words].append(next_word)
                else:
                    self.context[prev_words] = [next_word]

            # finally add the (</s>, </s>) -> <s> | (</s>, <s>) -> <s> ...
            final = ["</s>"] * (self.N - 1)
            for _ in range(self.N - 1):
                self.context[tuple(final)] = ["<s>"]
                final.pop(0)
                final.append("<s>")

        print(f"Time Taken to Create Model : {time.time() - now}")
```

Then we can put in our helper functions for generating text.

```

def next_rand_word(self, prev_words:tuple):
    return random.choice(self.context[prev_words])

def generate_text(self, n_words = 50):

    result = []
    prev = ["<s>"] * (self.N - 1)

    step = 0
    while step < n_words:
        next = self.next_rand_word(tuple(prev))

        if next not in ["<s>", "</s>"]:
            result.append(next)
            step += 1

        prev.pop(0)

        prev.append(next)

    print(" ".join(result))

```

Now running it on Mary Shelly's Frankenstein gives the following:

```

m = NgramModel(5, "/home/mbahng/Desktop/Chatbot/Frankenstein.txt")
m.generate_text(500)

```

Time Taken to Create Model : 0.5771486759185791

As I sat , a train of reflection occurred to me which led me to consider the effects of what I was now doing . Reserve on such a point would be not only useless , but draw down treble misery on us all. " I trembled violently at his exordium , and my father reposed . My abhorrence of this fiend can not be conceived . As I spoke , rage sparkled in my eyes ; the females were flying and the enraged Felix tearing me from his father , to whose knees I clung , in a transport

However, we end with some problems of this model. Remember that we must compute

$$\mathbb{P}(w_t \mid w_{t-1}, \dots, w_1) \approx \mathbb{P}(w_t \mid w_{t-1}, \dots, w_{t-N+1}) = \frac{\mathbb{P}(w_t, w_{t-1}, \dots, w_{t-N+1})}{\mathbb{P}(w_{t-1}, \dots, w_{t-N+1})}$$

at every time step.

1. One sparsity problem is that we will compute $\mathbb{P}(w_t, w_{t-1}, \dots, w_{t-N+1}) = 0$ since w_t never followed after the phrase $w_{t-N+1}, \dots, w_{t-1}$, which may be inaccurate if w is indeed used often outside the context. This can be fixed by **smoothing**, i.e. adding a small δ to the count for every $w \in \mathcal{V}$.
2. If the denominator itself doesn't occur at all in the text, then its probability will be 0! Therefore, we can't calculate the probability for *any* w_t ! What we can do is just condition on a shorter phrase with the following approximate, which is called **backoff**.

$$\mathbb{P}(w_t \mid \underbrace{w_{t-N+1}, \dots, w_{t-1}}_{N-1 \text{ words}}) \approx \mathbb{P}(w_t \mid \underbrace{w_{t-N+2}, \dots, w_{t-1}}_{N-2 \text{ words}})$$

3. It may require a lot of memory to store these giant dictionaries containing millions or billions of elements. This is not scalable.

Note that increasing N makes the sparsity problems worse, and typically we can't have N bigger than 5.

5.2 Fixed Window Neural Model

Neural language models have many advantages over the n-gram language. Compared to n-gram models, neural language models can handle much longer histories, can generalize better over contexts of similar words, and are more accurate at word prediction. On the other hand, neural net language models are much more complex, are slower and need more energy to train, and are less interpretable than n-gram models, so for many (especially smaller) tasks an n-gram language model is still the right tool.

A neural language model is similar to an n-gram as it takes as input at time t a representation of some number of previous words (w_{t-1}, w_{t-2}, \dots) and outputs a probability distribution of w_t over some possible set of words. We also estimate based on a constant number of N previous words:

$$\mathbb{P}(w_t \mid w_1, \dots, w_{t-1}) \approx \mathbb{P}(w_t \mid w_{t-N+1}, \dots, w_{t-1})$$

If you are familiar with neural nets, then with word embeddings, you should have a good idea of how this can be implemented. Another advantage of using embeddings is that neural nets can generalize better to unseen data. For example, if we have never seen the word “dog” but have seen “cat,” we can use the fact that the two words have similar embeddings to predict the next words after “dog,” whereas in a vanilla n-gram model we don’t know.

Now given a corpus with total vocabulary \mathcal{V} consisting of words v_i for $i = 1, \dots, |\mathcal{V}|$, we can one-hot encode all the vectors $\mathbf{v}_i \mapsto \mathbf{x}_i$, which are the standard vectors. We can train a d -dimensional embedding on the words (e.g. doc2vec) which results in a $d \times |\mathcal{V}|$ **embedding matrix** \mathbf{E} of form

$$\begin{bmatrix} | & \dots & | \\ \mathbf{E}_1 & \dots & \mathbf{E}_{|\mathcal{V}|} \\ | & \dots & | \end{bmatrix}$$

where the i th column \mathbf{E}_i represents the embedding of v_i . This makes it easy to map the one-hot vectors since $\mathbf{E}_i = \mathbf{E}\mathbf{x}_i$. Now, just like the n-gram model, we must use the previous N words to predict the next word. Therefore, if we have words v_{t-N}, \dots, v_{t-1} , we can calculate their embeddings and use these vectors to train our model. The forward pass of the multilayer perceptron is outlined below:

1. Given the N previous words $\mathbf{x}_{i_{t-N}}, \dots, \mathbf{x}_{i_{t-1}}$, take their embedding $\mathbf{E}\mathbf{x}_{i_{t-N}}$ and vertically stack them on top of each other to create a Nd -dimensional vector.

$$\mathbf{e} = [\mathbf{E}\mathbf{x}_{i_{t-N}}; \dots; \mathbf{E}\mathbf{x}_{i_{t-1}}]$$

2. We input this through a hidden layer by multiplying it by $\mathbf{W}^{[1]}$ (and adding a bias term) and then through an activation layer $\sigma^{[1]}$. This is done L times for a L -layer MLP:

$$\mathbf{h}_\theta(\mathbf{e}) = \sigma^{[L]} \circ \mathbf{W}^{[L]} \circ \sigma^{[L-1]} \circ \mathbf{W}^{[L-1]} \circ \dots \circ \sigma^{[1]} \circ \mathbf{W}^{[1]}(\mathbf{e})$$

3. We multiply by the final linear map $\mathbf{U} : \mathbb{R}^{N^{[L]}} \rightarrow \mathbb{R}^{|\mathcal{V}|}$ and then apply the softmax activation function to get a vector of probabilities.

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{U}\mathbf{h}_\theta(\mathbf{e}) + \mathbf{b}_\mathbf{U})$$

Even though this does eliminate all the sparsity problems of an n-gram model, there are some remaining problems.

1. First, while we can always increase the window size, it can never be large enough. That is, there may always be some dependencies that we must watch out for that is beyond the window length of the neural model.
2. Furthermore, words in different positions are multiplied by completely different weights in W , and so there is no symmetry in how the inputs are processed.

Therefore, we need a neural architecture that can process inputs of any length, i.e. a RNN.

5.3 Vanilla RNNs

The recurrent neural network can process any length input, the model size does not increase for longer input text, and the same weights are applied on every timestep, so there is symmetry in how inputs are processed. However, the computation can be slow since this is sequential by definition. We can take the corpus, add the START and END tokens in it, and train the RNN with minibatch SGD by sampling 32 to 64 sentences from the corpus and computing the gradient of the cross-entropy loss with teacher forcing. Then, we can simply input in the START token as the first input \mathbf{x}_1 , which will generate the next word that we set as the second sequential input $\hat{\mathbf{y}}_1 = \mathbf{x}_2$.

5.3.1 Bidirectional RNNs

For problems in, say sentiment classification, the context of not just the previous words, but also the future words, may be relevant to determining the sentiment. For example, take the following sentence:

The movie was terribly exciting.

The terribly part might be interpreted as a negative sentiment, when it is actually being used as an adjective to describe exciting. Therefore, we want to use bidirectional or multilayer RNNs to capture the entire sentence before outputting anything.

Note that bidirectional RNNs are only applicable if you have access to the entire input sequence. You can't use them in language modeling since in LM you only have *left* context available. If you do have the entire input sequence, bidirectionality is powerful and you should use it by default. For example, BERT (Bidirectional Encoder Representations from Transformers) is a powerful pretrained contextual representation system built on bidirectionality.

5.3.2 Perplexity Evaluation

The standard evaluation metric for language models is **perplexity**, which has a very nice intuitive definition behind it. Assume that a stream of words $\mathbf{x}_1, \dots, \mathbf{x}_T$ is generated by the model. We can calculate the probability of this sentence being generated by taking the product of conditional probabilities

$$\prod_{t=1}^T \mathbb{P}_{\text{LM}}(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \dots, \mathbf{x}_1)$$

This measure would mean that longer sentences would have less probability assigned to them just because they are longer, so we must take the geometric mean of this.

$$\left(\prod_{t=1}^T \mathbb{P}_{\text{LM}}(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \dots, \mathbf{x}_1) \right)^{1/T}$$

If this number is high, then this means that the probability of this sentence being generated is also high and therefore this language model is good. If we consider all the conditional distributions to be very sharply peaked around one word, then there is not much variability on how the sentence is generated. However, if all the conditional distributions are generally uniform, then there is a lot of variability in the sentence generation. In fact, this is closely related to the concept of entropy. In fact, a uniform conditional probability distribution for all possible cases would correspond to both maximum entropy and minimum probability since (if V is the size of the vocabulary)

$$\left(\prod_{t=1}^T \mathbb{P}_{\text{LM}}(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \dots, \mathbf{x}_1) \right)^{1/T} = \left(\prod_{t=1}^T \frac{1}{V} \right)^{1/T} = \frac{1}{V}$$

Finally, perplexity itself is just defined as the inverse of this probability, so lower perplexity is better.

$$\prod_{t=1}^T \left(\frac{1}{\mathbb{P}_{\text{LM}}(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \dots, \mathbf{x}_1)} \right)^{1/T}$$

It can also be seen as equivalent to the cross-entropy loss. Of course, perplexity is a stochastic variable, but on average, the worst possible perplexity is V itself, which may be very high if the corpus is large. Some common perplexities are shown below, with later rows representing increasingly complex RNNs.

perplexity.png

5.3.3 BLEU

The **Bilingual Evaluation Understudy (BLEU)** compares the machine-written translation to one or several human-written translations, and computes a similarity score based on

1. n-gram precision (usually for 1, 2, 3, 4-grams)
2. plus a penalty for too-short system translations.

It is useful but imperfect since there are many valid ways to translate a sentence. Therefore, a good translation can get a poor BLEU performance because it has low n-gram overlap with the human translation.

5.4 LSTMs

However, the problem of vanishing (and exploding) gradients render the vanilla RNN practically infeasible. Due to the exponential memory loss of the RNN, these models have a hard time modelling long-term dependencies between words. For example, take a look at the following sentence, with a blank at the end where the RNN-LM must fill out.

When she tried to print her tickets, she found that the printer was out of toner.
She went to the stationery store to buy more toner. It was very overpriced. After
installing the toner into the printer, she finally printed her _____.

In here, the RNN would have to remember the “tickets” on the 7th word all the way up to the end. Therefore, we introduce the LSTM model.

6 Encoder-Decoder Machine Translation Models

Let us first focus on the problem of language translation. **Machine Translation (MT)** is the task of translating a sentence x from one language (the **source language**) to a sentence y in another language (the **target language**).

```
x:  L'homme est ne libre, et partout il est dans les fers
y:  Man is born free, but everywhere he is in chains
```

6.1 Statistical Machine Translation

Machine translation first started in the 1950s when, following the computer’s extreme success in many problems, the U.S. decided to use them for translating Russian to English. This was a much harder problem to solve, but in the 1990s to 2010s, this idea was reborn again in the form of **statistical machine translation (SMT)**. The idea was to learn a probabilistic model from the data. That is, given a French sentence x , we want to find the best English sentence y , which we can use Bayes rule to break this down into two components to be learned separately.

$$\arg \max_y \mathbb{P}(y | x) = \arg \max_y \mathbb{P}(x | y) \mathbb{P}(y)$$

1. The left hand distribution $\mathbf{P}(x | y)$ models how words and phrases should be translated (*fidelity*), which are learned from parallel data (data of paired English/French sentences).
2. The right distribution $\mathbf{P}(y)$ models how to write good English (*fluency*), which is learnt from monolingual data.

This obviously requires a lot of parallel data, which can be found in international conferences and such. To go into the details, we want to learn the translation model $\mathbb{P}(x | y)$ from the parallel corpus by introducing a latent variable a into the model $\mathbf{P}(x, a | y)$, where a is called the **alignment**. This represents the word-level correspondence between source sentence x and target sentence y .

alignment.png

Alignment refers to the correspondence between particular sets of words in the translated sentence pair. Note that it is not as simple as just connecting two individual words, as some words may need to be translated into a phrase, vice versa, or some phrases/words may have no counterpart at all! Not to mention we also need to account for words with multiple meanings. From this, it is clear that alignment can be extremely complex. Essentially, the distributions are learned as a combination of many factors, including the probability of particular words aligning, and the alignments are also learned (they are not labeled in the data!) using special learning algorithms like Expectation-Maximization (EM).

During this time period, SMT was a huge field, with extremely complex SOTA systems with hundreds of details. These systems had many separately-designed subcomponents and required a lot of feature engineering (designing features to capture particular language phenomena). It also required compiling and maintaining extra resources, like tables or equivalent phrases, necessitating a lot of human effort.

6.2 Neural Machine Translation

Post-2010, **neural machine translation (NMT)** models, which performs machine translation with a single end-to-end neural network, have become SOTA. We can use a **sequence-to-sequence (seq2seq)** encoder-decoder model with some pretrained word embeddings to learn the translation. This seq2seq is an example of a conditional language model, since it is predicting the next word of the target sentence y by conditioning on the source sentence x . It directly calculates $\mathbb{P}(y | x)$ by expanding it as

$$\mathbb{P}(y | x) = \mathbb{P}(y_1 | x) \mathbb{P}(y_2 | y_1, x) \dots \mathbb{P}(y_T | y_1, \dots, y_{T-1}, x)$$

6.3 Attention

A huge issue with the sequence-to-sequence model is the **bottleneck problem**. The encoder encodes the input sentence into a single latent vector at the end, and this one vector needs to capture *all information* about the source sentence. Clearly, if this vector is not large enough, we have too little bandwidth to capture this information, resulting in an information bottleneck.

The idea of **attention** provides a solution to this bottleneck problem. Basically, we want to establish connections from the decoder to not just the last hidden state of the encoder, but to all of its nodes. Each encoder node represents some information about each word, and by taking some weighted sum of these nodes, we can choose which one to put this attention on. The specific steps are listed:

1. We have encoder hidden states $\mathbf{h}_1, \dots, \mathbf{h}_N \in \mathbb{R}^h$.
2. On timestep t , we have the decoder hidden state $\mathbf{s}_t \in \mathbb{R}^h$.
3. We get the attention scores \mathbf{e}^t for this step by computing

$$\mathbf{e}^t = (\mathbf{s}_t^T \mathbf{h}_1, \dots, \mathbf{s}_t^T \mathbf{h}_N) \in \mathbb{R}^N$$

4. We take its softmax to get the **attention distribution** $\boldsymbol{\alpha}^t$ for this step (a discrete probability distribution)

$$\boldsymbol{\alpha} = \text{softmax}(\mathbf{e}^t) \in \mathbb{R}^N$$

5. We use α^t to take a weighted sum of the encoder hidden states to get the attention output \mathbf{a}_t

$$\mathbf{a}_t = \sum_{i=1}^N \alpha_i^t \mathbf{h}_i \in \mathbb{R}^h$$

6. We concatenate the attention output \mathbf{a}_t with the decoder hidden state \mathbf{s}_t and proceed as in the non-attention seq2seq model.

$$[\mathbf{a}_t; \mathbf{s}_t] \in \mathbb{R}^{2h}$$

7 Transformer Based Models