CS4248 — NATURAL LANGUAGE PROCESSING

PROF. KAN MIN-YEN

arsatis[*]

CONTENTS

1 Introduction 4
- 1.1 Levels of Linguistic Knowledge . . . 4

2 Words 5
- 2.1 Regular Expressions . . . 5
- 2.2 Corpus Preprocessing . . . 6
- 2.3 Word Error Handling . . . 6
  - 2.3.1 Minimum Edit Distance . . . 7
  - 2.3.2 Noisy Channel Model . . . 7

3 Language Models 8
- 3.1 *n*grams . . . 8
- 3.2 Smoothing . . . 9
  - 3.2.1 Unknown Words . . . 9
  - 3.2.2 Add-*k* Smoothing . . . 10
  - 3.2.3 Backoff and Interpolation . . . 10
  - 3.2.4 Kneser-Ney Smoothing . . . 11
- 3.3 Evaluating Language Models . . . 11
  - 3.3.1 Perplexity . . . 12

4 Text Classification 12
- 4.1 Sentiment Analysis . . . 12
- 4.2 Naïve Bayes . . . 13

[*]author: https://github.com/arsatis

4.3 TF-IDF 14
4.4 Vector Space Model 15
4.5 Evaluating Classification 16

5 Machine Learning 17
5.1 Generative and Discriminative Classifiers 17
5.2 Logistic Regression 18
5.2.1 Cross Entropy 18
5.2.2 Stochastic Gradient Descent 19
5.3 Regularization 20
5.4 Neural Networks 20

6 Embeddings 21
6.1 Term-Context Matrices 21
6.2 Dense Vectors 22
6.3 Word2Vec 23
6.3.1 Skip-Gram Negative Sampling (SGNS) 25
6.3.2 Practical Considerations 26
6.4 Properties of Embeddings 27

7 Sequences 27
7.1 Parts of Speech 27
7.1.1 POS Tagging 28
7.2 Hidden Markov Models (HMM) 28
7.2.1 Forward Computation 29
7.2.2 Viterbi Decoding 29

8 Encoder-Decoder Models 30
8.1 Recurrent Neural Networks (RNN) 31
8.1.1 Conditional Language Models 31
8.2 Encoder-Decoder 32
8.3 Attention 34
8.4 Beam Search Decoding 35

9 Constituency Parsing 36
9.1 Context Free Grammars (CFGs) 37
9.2 Chomsky Normal Form (CNF) 38
9.3 Syntactic Parsing 39
9.4 Statistical Parsing 40

10 Semantics 41
10.1 Computational Lexical Semantics 41
10.1.1 WordNet 42
10.1.2 Word Similarity 43
10.2 Logical Semantics 44
10.2.1 First Order Logic 44

10.2.2 Lambda Calculus 45
10.2.3 Adjuncts and Roles 46

11 Classification Applications 46
11.1 Summarization 46
11.1.1 Summary Evaluation 47
11.1.2 Query-Based Summarization 48
11.2 Question Answering 49
11.2.1 IR-based QA Systems 50
11.2.2 Knowledge-Based QA Systems 52
11.2.3 Evaluation Metrics for QA 52

12 Sequence Applications 53
12.1 Machine Translation 53
12.2 Noisy Channel Model 54
12.2.1 Evaluation Metrics for MT 55

1 INTRODUCTION

Lecture 1
14th January 2022

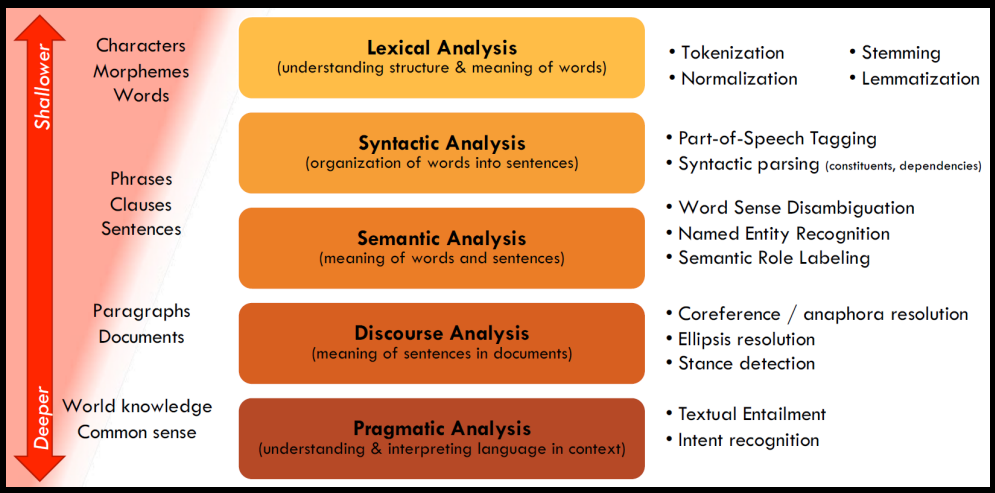

Figure 1: Outline of NLP.

NLP is complex due to the following factors:

- Ambiguity (i.e., one form can have different meanings)
- Sparsity (sparse data due to *Zipf's Law*)
- Variation (e.g., language used on social media vs in formal settings)
- Expressivity (i.e., the same meaning can be expressed with different forms)
- Unmodeled variables (e.g., Winograd Schemas)
- Unknown representations (we do not fully know how to represent the knowledge a human has or needs to understand language).

1.1 *Levels of Linguistic Knowledge*

Language consists of the following levels:

1. Character
2. Morpheme
3. Word
4. Clause
5. Sentence
6. Discourse
7. Corpus

2 WORDS

Lecture 2
21st January 2022

2.1 *Regular Expressions*

Regular expressions are a formal language for specifying a set of text strings. They can be considered as a pattern to specify text search queries to search a corpus of text. They consist of:

- Basic patterns:
  - `.`: matches any character except line breaks.
  - `^`: matches the start of a string.
  - `$`: matches the end of a string.
  - `|`: matches RE either before or after the symbol.
  - `\b`: matches boundary between word and non-word.
- Character classes: defines a set of valid characters.
  - Enclosed using `[...]`
  - Can be negated using `[^ ...]`
  - `\d`: matches any digit.
  - `\D`: matches any non-digit.
  - `\s`: matches any whitespace character.
  - `\S`: matches any non-whitespace character.
  - `\w`: matches any word character.
  - `\W`: matches any non-word character.
- Repetition patterns:
  - `+`: 1 or more occurrences.
  - `*`: 0 or more occurrences.
  - `?`: 0 or 1 occurrences.
  - $\{n\}$: exactly n occurrences.
  - $\{1, k\}$: between 1 and k occurrences; can be unbounded.
- Groups: organizing patterns into parts.
  - Enclosed using `(...)`
  - Can be **backreferenced** using $\backslash k$ when attempting to reference the k^{th} group.
- Lookarounds:
  - `(?=)`: positive lookahead.
  - `(?!)`: negative lookahead.
  - `(?<=)`: positive lookbehind.
  - `(?<!)`: negative lookbehind.

Note that a regex is an equivalent to an FSA.

2.2 *Corpus Preprocessing*

Preprocessing steps typically include:

- Tokenization: splitting strings into tokens (i.e., character sequences with a semantic meaning).
  - This can be done on the **character-level**, **subword-level**, or the **word-level**.
  - Some examples include:
    * SpaCy tokenizer
    * Maximum matching (for languages with no whitespace, e.g., Chinese/Japanese)
    * Byte-Pair Encoding (BPE), which consists of a 2-part setup:
      1. **Token Learning**
      2. **Token Segmenting**
- Normalization: converting text to a convenient, standard form. This can be done in various ways, e.g.:
  - using a predefined dictionary/list of equivalence classes, e.g., deleting periods in a term.
  - case folding
  - lemmatization: reducing inflections or variant forms to the base form.
    * An example is the Penn Treebank Tokenizer, which separates out clitics (e.g., doesn't → does + n't).
  - stemming: reducing terms to their stems, i.e., crude chopping of affixes.
    * An example is the Porter Stemmer.
- Segmentation: splitting a corpus into sentences.
  - Possible approaches:
    * Binary classification (i.e., end/not end of sentence) using:
      1. handwritten rules,
      2. set of RegEx, and/or
      3. machine learning.

2.3 *Word Error Handling*

Spelling error handling can be classified into 3 main categories:

- non-word error (i.e., misspellings) detection.
- isolated-word error correction.

- context-sensitive error detection and correction (i.e., use of context to detect and correct spelling errors).

2.3.1 Minimum Edit Distance

Replacements for these word errors can typically be determined using the Levenshtein/minimum edit distance heuristic. Some variants in the computation of minimum edit distance include:

- no transposition
- unweighted (i.e., insertion/deletion/substitution/transposition all costs 1).
- weighted (e.g., substitution costs 2)
- Needleman-Wunsch: gaps at the beginning and end of a sentence are not penalized.
- Smith-Waterman: ignore poorly aligning regions.

2.3.2 Noisy Channel Model

We initialize a corpus of annotated text, where misspelled words are identified and labelled with the correctly spelled ones. Then, for each misspelled word, we will gather the probability estimates, and guess the correct word w using the equation

$$\hat{w} = \arg\max_{w \in \mathrm{V}} \mathrm{P}(w|x) = \arg\max_{w \in \mathrm{V}} \mathrm{P}(x|w)\mathrm{P}(w)$$

- $\mathrm{P}(x|w)$ is almost impossible to predict; instead, we could estimate it based on simplifying assumptions, e.g.:
  - most misspelled typewritten words are single-errors $\implies$ only single-error misspellings are considered (e.g., insertion, deletion, substitution, transposition).
  - we can then define 4 *confusion matrices* (1 for each single-error type), where each confusion matrix lists the number of times one character was confused with another, e.g.:
  - subsequently, $\mathrm{P}(x|w)$ can be computed using the following equation:

$$\mathrm{P}(x|w) = \begin{cases} \frac{\mathrm{del}(w_{i-1},w_i)}{\mathrm{count}(w_{i-1},w_i)}, & \text{if deletion} \\ \frac{\mathrm{ins}(w_{i-1},w_i)}{\mathrm{count}(w_{i-1})}, & \text{if insertion} \\ \frac{\mathrm{sub}(x_i,w_i)}{\mathrm{count}(w_i)}, & \text{if substitution} \\ \frac{\mathrm{trans}(w_i,w_{i+1})}{\mathrm{count}(w_i,w_{i+1})}, & \text{if transposition} \end{cases}$$

sub[X, Y] = Substitution of X (incorrect) for Y (correct)

X \ Y (correct)	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	0	0	7	1	342	0	0	2	118	0	1	0	0	3	76	0	0	1	35	9	9	0	1	0	5	0
b	0	0	9	9	2	2	3	1	0	0	0	5	11	5	0	10	0	0	2	1	0	0	8	0	0	0
c	6	5	0	16	0	9	5	0	0	0	1	0	7	9	1	10	2	5	39	40	1	3	7	1	1	0
d	1	10	13	0	12	0	5	5	0	0	2	3	7	3	0	1	0	43	30	22	0	0	4	0	2	0
e	388	0	3	11	0	2	2	0	89	0	0	3	0	5	93	0	0	14	12	6	15	0	1	0	18	0
f	0	15	0	3	1	0	5	2	0	0	0	3	4	1	0	0	0	6	4	12	0	0	2	0	0	0
g	4	1	11	11	9	2	0	0	0	1	1	3	0	0	2	1	3	5	13	21	0	0	1	0	3	0
h	1	8	0	3	0	0	0	0	0	0	2	0	12	14	2	3	0	3	1	11	0	0	2	0	0	0
i	103	0	0	0	146	0	1	0	0	0	0	6	0	0	49	0	0	0	2	1	47	0	2	1	15	0
j	0	1	1	9	0	0	1	0	0	0	0	2	1	0	0	0	0	0	5	0	0	0	0	0	0	0
k	1	2	8	4	1	1	2	5	0	0	0	0	5	0	2	0	0	0	6	0	0	0	4	0	0	3
l	2	10	1	4	0	4	5	6	13	0	1	0	0	14	2	5	0	11	10	2	0	0	0	0	0	0
m	1	3	7	8	0	2	0	6	0	0	4	4	0	180	0	6	0	0	9	15	13	3	2	2	3	0
n	2	7	6	5	3	0	1	19	1	0	4	35	78	0	0	7	0	28	5	7	0	0	1	2	0	2
o	91	1	1	3	116	0	0	0	25	0	2	0	0	0	0	14	0	2	4	14	39	0	0	0	18	0
p	0	11	1	2	0	6	5	0	2	9	0	2	7	6	15	0	0	1	3	6	0	4	1	0	0	0
q	0	0	1	0	0	0	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r	0	14	0	30	12	2	2	8	2	0	5	8	4	20	1	14	0	0	12	22	4	0	0	1	0	0
s	11	8	27	33	35	4	0	1	0	1	0	27	0	6	1	7	0	14	0	15	0	0	5	3	20	1
t	3	4	9	42	7	5	19	5	0	1	0	14	9	5	5	6	0	11	37	0	0	2	19	0	7	6
u	20	0	0	0	44	0	0	0	64	0	0	0	0	2	43	0	0	4	0	0	0	0	2	0	8	0
v	0	0	7	0	0	3	0	0	0	0	0	1	0	0	1	0	0	0	8	3	0	0	0	0	0	0
w	2	2	1	0	1	0	0	2	0	0	1	0	0	0	0	7	0	6	3	3	1	0	0	0	0	0
x	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0
y	0	0	2	0	15	0	1	7	15	0	0	0	2	0	6	1	0	7	36	8	5	0	0	1	0	0
z	0	0	0	7	0	0	0	0	0	0	0	7	5	0	0	0	0	2	21	3	0	0	0	0	3	0

Source: A Spelling Correction Program Based on a Noisy Channel Model (Kernighan et al., 1990)

Figure 2: Confusion matrix for substitution errors.

- $P(w)$ can be estimated using a **Maximum Likelihood Estimation (MLE)**, i.e.:

$$P(w) = \frac{\text{freq}(w)}{N}$$

However, the limitation of the noisy channel model is that there is *no consideration of the additional context*, thus the model is only applicable for *non-word errors*.

3 LANGUAGE MODELS

Lecture 3
28th January 2022

Language models are models that assign probabilities to a sentence. They can be used to estimate the probability of a sequence of words (i.e., $P(W) = P(w_1, \ldots, w_n)$), or the probability of an upcoming word (i.e., $P(w_k|w_1, \ldots, w_{k-1})$).

3.1 *ngrams*

To estimate the probability of a sequence of words $P(w_1, \ldots, w_n)$, we could adopt the chain rule to get:

$$P(w_1, \ldots, w_n) = P(w_1) \times P(w_2|w_1) \times \ldots \times P(w_n|w_1, \ldots, w_{n-1})$$

$$= \prod_{i=1}^{n} P(w_i|w_1, \ldots, w_{i-1}) \quad (1)$$

In fact, we could adopt the **Markov assumption**, i.e., approximate the probability by assuming that it is just dependent on the last $(k-1)$ words. Therefore, equation 1 would simplify to

$$\mathrm{P}(w_1, \ldots, w_n) = \begin{cases} \prod_{i=1}^{n} \mathrm{P}(w_i) & \text{if } k = 1 \\ \prod_{i=1}^{n} \mathrm{P}(w_i | w_{i-k+1}, \ldots, w_{i-1}) & \text{if } k \geq 2 \end{cases}$$

To estimate the probability of an upcoming word, we could use **Maximum Likelihood Estimation (MLE)**, where the probabilities are estimated by getting *counts* from the corpus and *normalizing* by the sum of all n-grams that share the preceding words, i.e.:

$$\mathrm{P}_{\mathrm{MLE}}(w_i | w_{i-k+1}, \ldots, w_{i-1}) = \frac{\mathrm{count}(w_{i-k+1}, \ldots, w_{i-1}, w_i)}{\mathrm{count}(w_{i-k+1}, \ldots, w_{i-1})} \tag{2}$$

where $k(\geq 2)$ represents the size of the tokens/grams used by the language model.

Since multiplying MLE probabilities could result in underflow, they are often *converted to logarithms* for the purpose of comparisons. This is a valid conversion because logarithms are strictly *monotonic*, and so

$$\prod_i \mathrm{P}_i \propto \sum_i \log \mathrm{P}_i$$

3.2 *Smoothing*

3.2.1 Unknown Words

One way in which language models can be classified is by their vocabulary.

- **Open vocabulary**: the test set may contain words that are not in the vocabulary; in such cases, *OOV (out of vocabulary)* markers may be used.
- **Closed vocabulary**: the vocabulary is fixed, and all datasets contain only words from this vocabulary.

Open vocabulary models may result in some unseen words being attributed a count of 0 (and therefore a probability of 0), which could result in inappropriate word/sentence likelihood computations. To address this issue, smoothing techniques may be applied.

3.2.2 Add-k Smoothing

Assume that we are using a bigram model. In Laplace (or add-1) smoothing, we simply add 1 to all counts in our vocabulary. Therefore, the probability will be

$$\begin{aligned} \mathrm{P}_{\text{laplace}}(w_n|w_{n-1}) &= \frac{\mathrm{C}_{\text{laplace}}(w_{n-1}, w_n)}{\sum_w \mathrm{C}_{\text{laplace}}(w_{n-1}w)} \\ &= \frac{\mathrm{C}(w_{n-1}, w_n) + 1}{\mathrm{C}(w_{n-1}) + \mathrm{V}} \end{aligned}$$

where V is the size of the vocabulary. However, we observe that for some n-grams, too much weight gets shifted to the zero probabilities.

To assess the effect of smoothing on counts, we can define the **discounted $n$gram count**,

$$\mathrm{C}^*(w_{n-1}, w_n) = \mathrm{C}(w_{n-1}) \times \frac{\mathrm{C}(w_{n-1}, w_n) + 1}{\mathrm{C}(w_{n-1}) + \mathrm{V}} \tag{3}$$

and the **discount**, i.e., the ratio of the discounted counts to the original ones:

$$d_c = \frac{\mathrm{C}^*(w_{n-1}, w_n)}{\mathrm{C}(w_{n-1}, w_n)} \tag{4}$$

Note that the Laplace discount d_c is only defined where original counts > 1. In general, bigger counts are "penalized" more than smaller counts when add-1 smoothing is applied.

Add-k smoothing is a simple generalization of Laplace smoothing, where k is added to all counts in our vocabulary. This gives us the following equation:

$$\mathrm{P}_{add-k}(w_n|w_{n-1}) = \frac{\mathrm{C}(w_{n-1}, w_n) + k}{\mathrm{C}(w_{n-1}) + k\mathrm{V}} \tag{5}$$

3.2.3 Backoff and Interpolation

The underlying intuition of backoff and interpolation is that using less context may be a good thing. Therefore, we can mix the probability estimates from all the n-gram estimators, e.g., weighing and combining n-gram, $(n-1)$-gram, ..., and unigram counts:

$$\begin{aligned} P_{B\&I}(w_n|w_1, w_2, \ldots, w_{n-1}) =& \lambda_1 P(w_n|w_1, w_2, \ldots, w_{n-1}) + \lambda_2 P(w_n|w_2, \ldots, w_{n-1}) \\ &+ \ldots + \lambda_{n-1} P(w_n|w_{n-1}) + \lambda_n P(w_n) \end{aligned} \tag{6}$$

where $\sum \lambda_i = 1$, and each λ_i is estimated through parameter tuning.

3.2.4 Kneser-Ney Smoothing

Kneser-Ney smoothing is given by the equation

$$P_{KN}(w_n|w_{n-1}) = \frac{\max\left\{C(w_{n-1}, w_n) - \delta, 0\right\}}{C(w_{n-1})} + \lambda(w_{n-1}) P_{KN}(w_n) \tag{7}$$

intuitively, $P_{KN}(w_n)$ represents how likely is w_n to appear as a continuation of other words.

where

$$P_{KN}(w_n) = \frac{\left|\left\{w : C(w, w_n) > 0\right\}\right|}{\sum_{w'} \left|\left\{v : C(v, w') > 0\right\}\right|}$$

note that the cardinalities count the number of n-grams (w, w_n), and not the total frequency of these n-grams.

and

$$\lambda(w_{n-1}) = \frac{\delta}{C(w_{n-1})} \times \left|\left\{w' : C(w_{n-1}, w') > 0\right\}\right|$$

intuitively, $\lambda(w_{n-1})$ is a normalizing factor that distributes the discounts, and the cardinality function represents the number of words that can follow w_{n-1} (i.e., number of times the normalized discount has been applied).

There are two main features in Kneser-Ney smoothing, namely:

1. Absolute discounting: a fixed value δ is removed from bigram counts, where typically $0 < \delta < 1$.
    - The intuition behind this is that large counts would not be affected significantly, whereas smaller counts (which are not very useful) would be affected more.
2. Interpolation: discounts are distributed to the unseen n-grams.

3.3 *Evaluating Language Models*

Two main ways of evaluating a language model include:

- **intrinsic evaluation**: uses an *intrinsic metric* to evaluate the model (e.g., perplexity); this usually involves 3 steps:
    1. training the model on a **training set**.
    2. tuning the parameters on a **development set**.

3. testing the model on a **test set**, and using an evaluation metric to assess model performance.

- **extrinsic evaluation**: requires a *downstream task* (e.g., comparing the results of two LMs by running the output of both through a speech recognizer).

3.3.1 Perplexity

One metric that can be used to evaluate language models is **perplexity** PP(W), which is the inverse probability of the test set W, normalized by the number of words. For a test set $W = w_1, \ldots, w_n$,

$$\begin{aligned} PP(W) &= P(w_1, \ldots, w_n)^{-\frac{1}{n}} \\ &= \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_1, \ldots, w_{i-1})}} \\ &= \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_{i-1})}} \qquad \text{for a bigram model} \end{aligned} \tag{8}$$

Typically, perplexity is high when there is a *distributional mismatch*, i.e., when there are frequent n-grams in training but rare n-grams in the test set, and vice versa.

Perplexity can also be thought of as the number of possible next units (words) that can follow any unit (word).

4 TEXT CLASSIFICATION

Lecture 4
4th February 2022

Text classification is a task where given a document $x = (w_1, \ldots, w_{|x|}) \in V$, we are required to predict a label $y \in Y$, where Y is an enumerated, fixed set of classes.

Let $h(x)$ be the true mapping from input data $x \in X$ to some label(s) $y \in Y$; we can find an approximation $\hat{h}(x)$ to $h(x)$ using either a *rule based* approach, or using *supervised learning*.

4.1 *Sentiment Analysis*

Sentiment analysis is an application of text classification, where we are required to return the sentiment expressed in a text (i.e., author's subjective or

emotional attitude towards the central topic of the text). Some factors which we have to consider when dealing with sentiment analysis tasks include:

- indicator words (e.g., "like", "amazing", "hate", etc.)
- negations (e.g., "worst" vs "not the worst")
- context
- size of training data

4.2 *Naïve Bayes*

Naïve Bayes is a simple classification method based on Bayes' rule, which relies on a **Bag of Words (BoW)** representation of documents. Recall that in Bayes' rule:

$$\mathrm{P}(y|w) = \frac{\mathrm{P}(w|y)\mathrm{P}(y)}{\mathrm{P}(w)} \tag{9}$$

- $\mathrm{P}(y|w)$ represents the **posterior**, i.e., how probable is the instance classified as a member of class y?
- $\mathrm{P}(y)$ represents the **prior**, i.e., how probable is a document a member of y, without seeing any data?
- $\mathrm{P}(w|y)$ represents the **likelihood**, i.e., how probable is the data given that our document is a member of y?
- $\mathrm{P}(w)$ represents the **marginal**, i.e., how probable is the evidence under any class?

More specifically, the equation for Naïve Bayes is given as

$$y_{\mathrm{NB}} = \arg\max_{y \in \mathrm{Y}} \mathrm{P}(y|w) = \arg\max_{y \in \mathrm{Y}} \left(\prod_{i=1}^{n} \mathrm{P}(w_i|y) \right) \mathrm{P}(y) \tag{10}$$

where the following assumptions are made:

1. **Bag of words**: position of the word in the sentence doesn't matter.
2. **Conditional independence**: there are no interactions/relationships between words.

and the parameters $\mathrm{P}(w_i|y)$ and $\mathrm{P}(y)$ can be estimated by:

- $\hat{\mathrm{P}}(w_i|y) = \frac{\mathrm{C}(w_i,y)}{\sum_{w \in \mathrm{V}} \mathrm{C}(w,y)}$, i.e., fraction of times word w_i appears among all words in all documents on topic y.

- $\hat{P}(y) = \frac{N_y}{N}$, where N_y represents the number of documents belonging to class y, and N represents the total number of documents.

As a (class-specific) language model, Naïve Bayes encounters similar issues as other language models, including:

1. **Sparsity/Numerical underflow**.
   - Solution: we could transform to the equivalent addition of log probabilities.
2. **Out of vocabulary words**.
   - Solution: smoothing, backoff and interpolation, subword tokenizing, etc.

However, it is typically more robust, trains faster, and has lower storage requirements.

TrainNaiveBayes ($\mathcal{D}$, $\mathcal{Y}$) **returns** $\log P(y)$ and $\log P(w|y)$:
$N = D$
for each class $y \in \mathcal{Y}$: // Calc prior terms
  N[y] = D_y
  logprior[y] ← log N[y]/N
  V ← vocabulary of $\mathcal{D}$
  bigdoc[y] ← append(d) **forall** d in N[y]:
  for each word $w \in V$: // *Calc likelihood terms*
    $c(w, y)$ ← # of occurrences of w in bigdoc[y]
    loglikelihood[w,y] ← $\log \frac{c(w,y)}{\sum_{w' \in V}(c(w',y)+1)}$
return *logprior*, *loglikelihood*, V

TestNaiveBayes ($\boldsymbol{x}$, logprior, loglikelihood, $\mathcal{Y}$, V) returns y:
for each class $y \in \mathcal{Y}$:
  sum[y] ← logprior[y]
  for each position i in $\boldsymbol{x}$
    $w \leftarrow x_i$
    if $w \in V$:
      sum[y] += loglikelihood[w,c]
return $argmax_y$(sum[y])

Figure 3: Algorithm for Naïve Bayes.

4.3 *TF-IDF*

- **Count vectors** are vectors which store the number of occurrences of each term in a document.
- The **term frequency** ($tf_{w,d}$) of word w in document d is defined as the number of times that w occurs in d.
  - Since relevance does not increase proportionally with term frequency, we do not want to use raw term frequency.
  - Instead, we define the **log frequency weight** of word w in document d as

$$w_{t,d} = \begin{cases} 1 + \log_{10} tf_{t,d} & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

- The **document frequency** (df_w) of w is defined as the number of documents that contain w.
  - Intuitively, rare terms are more informative than frequent terms; so df_w is an inverse measure of the informativeness of w.
  - We define the **inverse document frequency** (idf_w) of w by

$$idf_w = \log_{10} \frac{\mathrm{N}}{df_w}$$

    where N is the total number of documents in a collection.

Together, the tf-idf weight of a term is given as

$$tfidf_{w,d} = (1 + \log_{10} tf_{w,d}) \times \log_{10}\left(\frac{\mathrm{N}}{df_w}\right) \tag{11}$$

The value of tf-idf increases with:

- the number of occurrences of the word within a document, and
- the rarity of the word in the collection.

4.4 *Vector Space Model*

We can represent documents in a V-dimensional vector space, where:

- words are axes of the space, and
- documents are points/vectors in the space.

However, this vector representation of documents is very sparse.

To determine the similarity between two documents, we could utilize the **cosine function**, which is a normalization of the dot product function:

$$\cos(v, w) = \frac{v \cdot w}{|v||w|} = \frac{\sum_{i=1}^{n} v_i w_i}{\sqrt{\sum_{i=1}^{n} v_i^2}\sqrt{\sum_{i=1}^{n} w_i^2}} \tag{12}$$

Since raw frequency values are non-negative, the cosine for the term-term matrix vectors range from 0 to 1.

4.5 *Evaluating Classification*

For any set of labels, there are 2 ways to be wrong and 2 ways to be right.

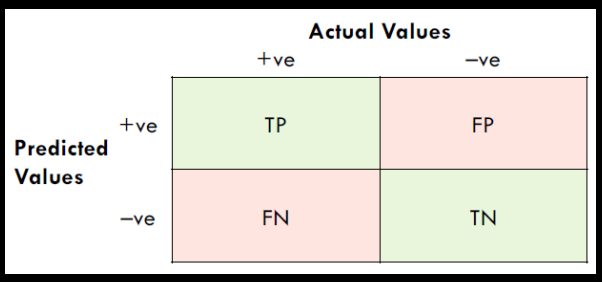

Figure 4: The binary confusion matrix.

- **False positive (FP, Type I)**: the system incorrectly predicts the label.
- **False negative (FN, Type II)**: the system incorrectly fails to predict the label.
- **True positive (TP)**: the system correctly predicts the label.
- **True negative (TN)**: the system correctly predicts that the label does not apply.

There are several metrics which could be used to evaluate the classification by a language model include:

- **accuracy**:

$$acc(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} [y^{(i)} = \hat{y}] \tag{13}$$

- **precision**: the fraction of *positive* predictions that are correct.

$$p = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{14}$$

in most systems, precision decreases when the number of inputs assigned to a class increases (i.e., when recall increases)

- **recall**: the fraction of *positive* instances which were correctly classified.

$$r = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{15}$$

- F **score**: a *harmonic mean* of precision and recall.

$$\text{F} = \frac{2}{\frac{1}{p} + \frac{1}{r}} = \frac{2pr}{p + r} \tag{16}$$

– We can also use the F_β **score**, where β is chosen such that recall is considered β times as important as precision:

$$F_\beta = \frac{(1+\beta^2)p \times r}{\beta^2 p + r}$$

When there are multiple classes, multiclass evaluation could come in the form of either:

- **micro averaging**: i.e., averaging over all TP, FP, FN, and TN.

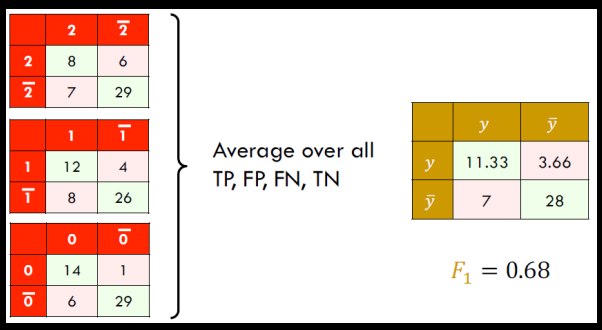

Figure 5: Example of micro averaging.

- **macro averaging**: i.e., averaging over all metrics.

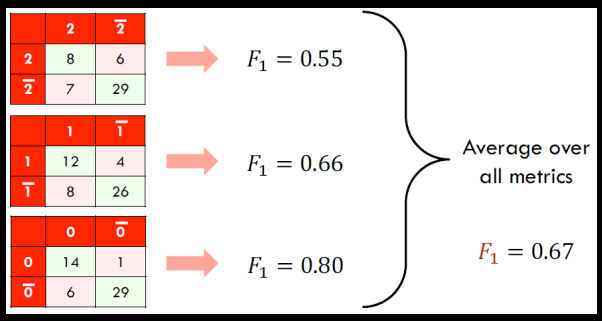

Figure 6: Example of macro averaging.

5 MACHINE LEARNING

Lecture 5
11th February 2022

5.1 *Generative and Discriminative Classifiers*

In general, a discriminative classifier (e.g., logistic regression) models the decision boundary between the classes, while a generative classifier (e.g., Naïve Bayes) models the actual distribution of each class. Specifically, a generative classifier learns the joint probability distribution $P(x, y)$ and predicts the

conditional probability with the help of Bayes' rule; a discriminative classifier, on the other hand, learns the conditional probability distribution $P(y|x)$.

For example, suppose we are distinguishing cat from dog images. Intuitively, a generative classifier would build a model of cat images and a model for dog images; during testing, the test images will be passed into both models and classified based on their fit to each model. On the other hand, a discriminative classifier simply distinguishes dogs from cats based on a single characteristic (e.g., the presence of collars in the image).

5.2 *Logistic Regression*

Logistic regression involves two phases:

- during the **training phase**, we learn weights θ and bias b using **stochastic gradient descent** and **cross-entropy loss**.
- during the **test phase**, when given a test example x, we compute $P(y|x)$ using the learned weights θ and bias b, and return whichever label (i.e., $y = 0$ or $y = 1$) has higher probability.
  - We will make use of the sigmoid function to determine the probability of each label, i.e.,

$$P(y = 1) = \sigma(\theta^T x) = \frac{1}{1 + e^{-(\theta^T x)}}$$

$$P(y = 0) = 1 - P(y = 1)$$

We can generalize logistic regression to 2 or more classes, by:

- generalizing to *multinomial logistic regression* (a.k.a. *maximum entropy modelling (MaxEnt)*.
- giving features separate weights for each of the k classes.
- upgrading sigmoid to **softmax**, i.e.,

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

  while keeping $\mathbb{R} \to [0, 1]$.

5.2.1 Cross Entropy

In supervised learning/classification, we are aware of the correct label y for each x, and we would like to set θ to *minimize the difference* between

our estimate $\hat{y}$ and the true y. To do this, we need a metric (i.e., a **loss/cost function**) and an *optimization algorithm* to update θ to minimize the loss.

Cross-entropy loss (a.k.a. negative log likelihood loss) is a case of conditional MLE, where we choose the parameters θ that maximize the log probability of the true y labels in the training data, given the observations x. In other words, we want to maximize

$$\begin{aligned} \mathrm{P}(y|x) &= \hat{y}^{y}(1-\hat{y})^{1-y} \\ \Leftrightarrow \log \mathrm{P}(y|x) &= y \log \hat{y} + (1-y)\log(1-\hat{y}) \end{aligned}$$

This is equivalent to minimizing

$$\mathrm{L}_{ce}(\hat{y}, y) = -\log \mathrm{P}(y|x) = -\Big(y \log \hat{y} + (1-y)\log(1-\hat{y})\Big) \tag{17}$$

for multi-class classification,

$$\mathrm{L}_{ce}(\hat{\theta}, z_i) = -\log \operatorname{softmax}(z_i)$$

where $\hat{\theta}$ is the vector output of the model and z_i is the true class.

5.2.2 Stochastic Gradient Descent

The idea of **gradient descent** is to find the gradient of the loss function at the current point, and move in the opposite direction of the gradient (i.e., direction of steepest descent).

Specifically, we start at $\theta(t)$ (i.e., settings of θ at time $t = 0$), and iteratively take steps (whose size depends on the magnitude of the learning rate α) along the direction with the steepest gradient:

$$\theta(t+1) = \theta(t) - \alpha \nabla_{\theta} \mathrm{L}\Big(h(x;\theta), y\Big) \tag{18}$$

The algorithm is given below in figure 7.

1. Initialize the weights at $t = 0$ to $\boldsymbol{\theta}(0)$
2. Do
3. Compute the gradient
$$\nabla(t) = \nabla L_{ce}(\boldsymbol{\theta}(t)) = \nabla\left(-\frac{1}{m}\sum_{j=1}^{m}\frac{y^{(j)}\boldsymbol{x}^{(j)}}{1+e^{y^{(j)}\boldsymbol{\theta}(t)\cdot x^{(j)}}}\right)$$
4. // Move in the direction $-\nabla(t)$
Update the weights $\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \alpha \nabla L_{ce}$
5. Continue to next iteration, until it is time to stop.
6. Return the final weights $\boldsymbol{\theta}^*$

Figure 7: Algorithm for logistic regression.

The termination condition can be set to either of the following criteria:

1. error **change** is small, and/or
2. error is small, and/or
3. maximum number of iterations is reached.

Instead of performing gradient descent, we could also opt for **stochastic gradient descent**, which chooses a single random example at a time, and the gradient is usually computed over batches of training instances, e.g.:

- entire dataset with m examples (i.e., **batch training**).
- k examples (i.e., **mini-batch training**).

5.3 *Regularization*

A model may **overfit** to the training data, and fail to **generalize** to examples in the test set. One solution is to add an overfit penalty $\Omega(\theta)$ to the loss function, such that

$$\hat{\theta} = \arg\max \sum_{j=1}^{m} \log \mathrm{P}(y^{(j)}|x^{(j)}) - \Omega(\theta)$$

There are two common variants for $\Omega(\theta)$, namely:

- **L2 (ridge) regularization**:

$$\hat{\theta} = \arg\max \sum_{j=1}^{m} \log \mathrm{P}(y^{(j)}|x^{(j)}) - \sum_{i=1}^{n} \theta_i^2$$

- **L1 (lasso) regularization**:

$$\hat{\theta} = \arg\max \sum_{j=1}^{m} \log \mathrm{P}(y^{(j)}|x^{(j)}) - \sum_{i=1}^{n} |\theta_i|$$

These choices of $\Omega(\theta)$ penalize large weights/classes of features.

5.4 *Neural Networks*

Even though logistic regression can represent any linear combination of features, it is incapable of representing nonlinear relationships between features.

One way of getting around this is by using **stacked linear/logistic regression**, i.e., the **ensemble** technique of **stacking**.

ensemble: combining 2 or more classifiers

stacking: using the output of one classifier as the input to another

Another way is to use **neural networks**, which operate via a **feedforward** (i.e., no loops) mechanism.

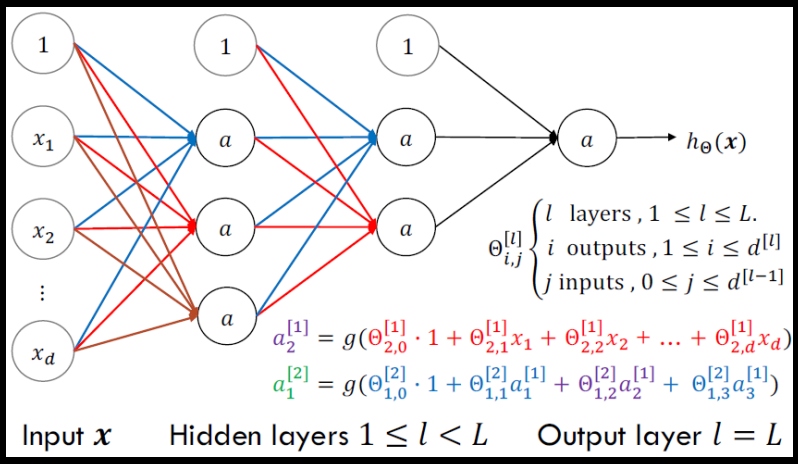

Figure 8: Example of a neural network. $\Theta_{output,input}^{[layer]}$ *represents indices of the network, whereas* $a_{index}^{[layer]}$ *represents activations of the network.*

The *final layer* utilizes a normalizing function (e.g., sigmoid, tanh, etc.), whereas the *intermediate layers* use **Rectified Linear Units (ReLU)**.

6 EMBEDDINGS

Lecture 6
18th February 2022

Most NLP algorithms require numerical and standardized input (e.g., vectors). So far, we have explored the **vector space model (VSM)**, where documents are represented using vectors, such as:

- **one-hot vectors**: vectors of length V, where a value of 1 in a dimension indicates that the document contains the word, and 0 otherwise.
- **TF-IDF vectors**: vectors of length V with the TF-IDF values of each term in the document.

However, these vectors have *no notion of similarity* between words (i.e., different words → orthogonal word vectors).

6.1 *Term-Context Matrices*

One solution to this is to use a **term-context matrix**, which captures the *distributional hypothesis* (i.e., if A and B have almost identical environments, we say that they are synonyms).

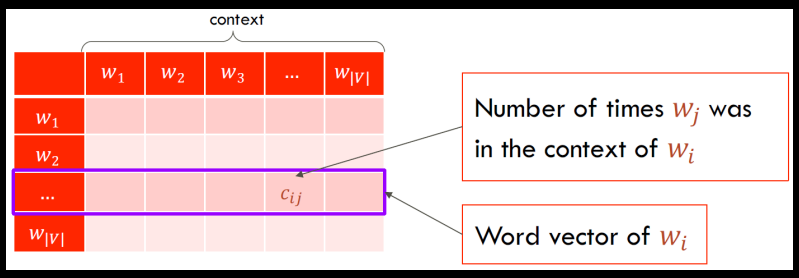

Figure 9: A term-context matrix.

possible definitions of "context" include the paragraph containing the term, or a window of n (e.g., 10) words

In term-context matrices, the entries of the matrix indicates the number of times a term is in another term's context. Thus, a word is now defined by a vector over counts of its context words, and we can define word similarity by comparing word vectors of the matrix. However, using raw counts may not be feasible, since certain words (e.g., stopwords) may be very frequent but not discriminative.

An alternative is to use the **Pointwise Mutual Information (PMI)** value, which checks if two words co-occur more than if they were independent. Formally, PMI is defined as

$$\text{PMI}(w_i, w_j) = \log \frac{\text{P}(w_i, w_j)}{\text{P}(w_i)\text{P}(w_j)}$$

However, since PMI could be negative, it is often thresholded, and positive PMI (PPMI) is used instead.

$$\text{PPMI}(w_i, w_j) = \max\left(\log \frac{\text{P}(w_i, w_j)}{\text{P}(w_i)\text{P}(w_j)}, 0\right) \tag{19}$$

Some other issues with PPMI/term-context matrices include:

- Handling rare words (i.e., words with small/zero counts)
  - Smoothing techniques (e.g., add-k, Kneser-Ney) can be applied.
  - Context probabilities can be raised (e.g., by increasing the size of the context).
- Sparsity: the matrix is of size $|\text{V}| \times |\text{V}|$, with most of the entries being 0.

6.2 *Dense Vectors*

Dense vectors have many practical benefits, including:

- less weights to tune

- lower risk of overfitting
- tendency to generalize better
- tendency to better capture synonymy

Optimally, we would like to be able to embed dense vectors in a manner such that words with similar contexts would have similar embeddings. Some approaches which could be used for such embeddings include:

- Singular value decomposition (SVD; matrix factorization)*
- Brown clustering*
- Neural network methods, e.g., Word2Vec.

6.3 *Word2Vec*

Word2Vec encompasses 2 network architectures: **CBOW** and **skip-gram**. Its underlying idea is to *employ a neural network to predict the next word*.

Basic setup:

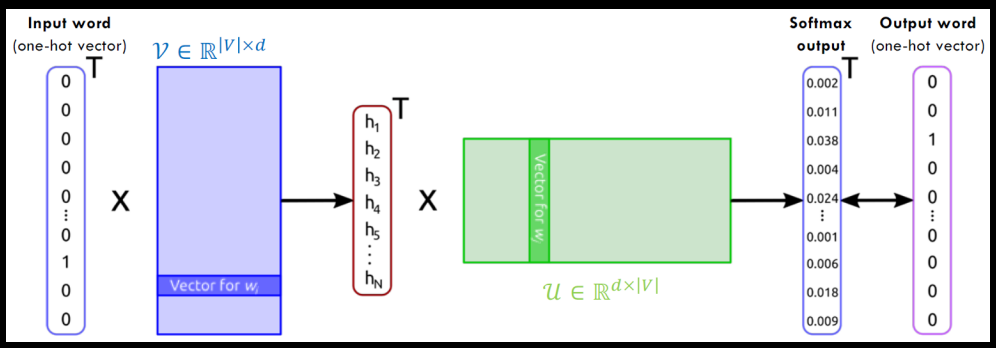

Figure 10: Setup of Word2Vec.

- We define two matrices:
  - Input embedding matrix: $\mathrm{V} \in \mathbb{R}^{|\mathrm{V}| \times d}$
  - Output embedding matrix: $\mathrm{U} \in \mathbb{R}^{d \times |\mathrm{V}|}$
  - Given a word w_i, let $v_i \in \mathrm{V}$ and $u_i \in \mathrm{U}$ be the input and output embeddings of w_i respectively.

Word2Vec learns 2 embeddings (i.e., u_i and v_i) for each word w_i

- Prediction task: given an input (one-hot) word vector w_i, what is the most likely output word?
  - $w_i^{\mathrm{T}} \cdot \mathrm{V} \implies v_i$
  - $\forall w \in \mathrm{V}, \mathrm{softmax}(v_i^{\mathrm{T}} \cdot \mathrm{U}) \implies \mathrm{P}(w|w_i)$

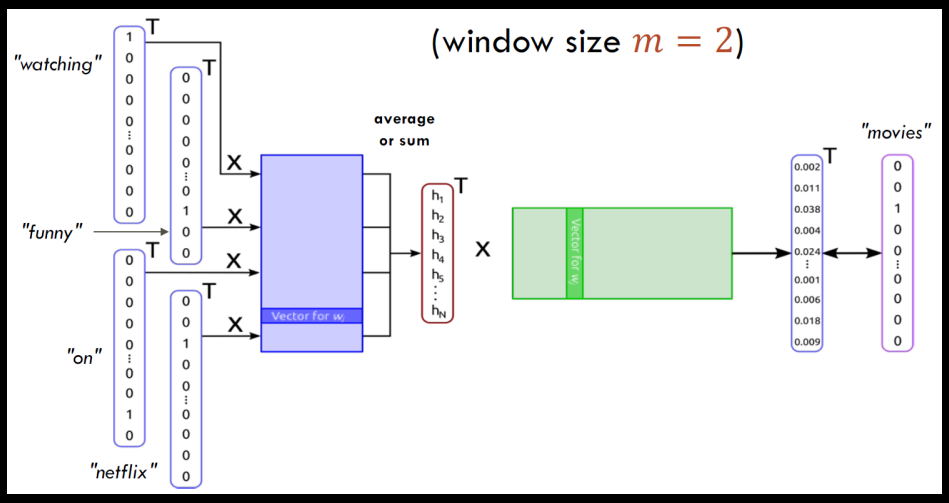

Figure 11: Example of a CBOW structure with window (i.e., context) size = 2.

For CBOW, the task is to predict a word given the surrounding context. The training objective is to minimize the loss function, given by

$$\begin{aligned} \mathrm{L} &= -\log \mathrm{P}(w_c | w_{c-m}, \ldots, w_{c+m}) \\ &= -\log \mathrm{P}(u_c | \tilde{v}) \\ &= -\log \frac{\exp(u_c^{\mathrm{T}} \cdot \tilde{v})}{\sum_{j=1}^{|\mathrm{V}|} \exp(u_j^{\mathrm{T}} \cdot \tilde{v})} \end{aligned} \tag{20}$$

where

$$\tilde{v} = \sum_{\substack{-m \le j \le m \\ j \neq 0}} v_{c+j}$$

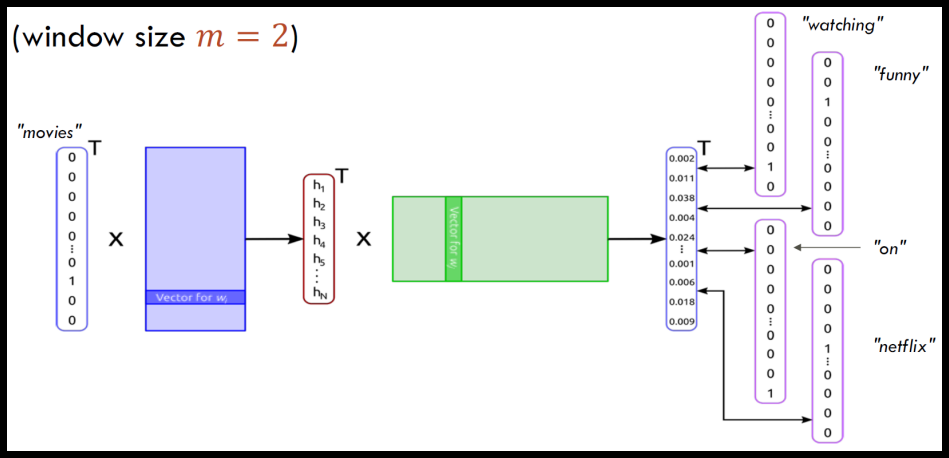

Figure 12: Example of a skip-gram structure with window size = 2.

On the other hand, the task for skip-grams is to predict the context of a given word. The loss function for skip-gram is given by

$$\begin{aligned} \mathrm{L} &= -\log \mathrm{P}(w_{c-m}, \ldots, w_{c+m} | w_c) \\ &= -\sum_{\substack{-m \le j \le m \\ j \neq 0}} \log \mathrm{P}(u_{c+j} | v_c) \\ &= -\sum_{\substack{-m \le j \le m \\ j \neq 0}} \log \left(\frac{\exp(u_{c+j}^{\mathrm{T}} \cdot v_c)}{\sum_{k=1}^{|\mathrm{V}|} \exp(u_k^{\mathrm{T}} \cdot v_c)} \right) \end{aligned} \tag{21}$$

The goal of Word2Vec is to maximize $(u_i^{\mathrm{T}} \cdot v_j)$, i.e., to ensure that the vectors/embeddings of words with similar contexts will be close to each other. The learning of U and V can be done via loss minimization using (stochastic) gradient descent, and the final embeddings can be chosen from either U or V, or an average of both matrices.

6.3.1 Skip-Gram Negative Sampling (SGNS)

There were several discrepancies between the textbook and the lecture slides for this section, so the equations/formula may be incorrect.

Notice that the loss function (i.e., in equations 20 and 21) involves the softmax function, which is rather expensive to compute (especially the denominator, which requires summing over the entire vocabulary). To address this, we can perform two tweaks to the algorithm:

- **Efficiency**: we subsample a smaller batch of weights to update, instead of updating all weights.
- **Effectiveness**: we pick informative samples (from both positive and negative samples) more often than uninformative ones.
- Thus, we will introduce two hyperparameters:
  - k: how many (more times of) negative samples to use?
  - α: which samples to use?

Therefore, instead of maximizing

$$\sigma(c \cdot m) = \frac{1}{1 + \exp(-c \cdot m)}$$

where c, m are the context and target words respectively, we could apply contrastive training (i.e., using both positive and negative samples) and minimize

the negative log likelihood

$$
\begin{aligned}
\mathrm{L} &= -\log\left(\prod_{(c,m)\in \mathrm{B}_+} \mathrm{P}(+|c,m) \prod_{(c,m)\in \mathrm{B}_-} \mathrm{P}(-|c,m)\right) \\
&= -\left(\sum_{(c,m)\in \mathrm{B}_+} \log \mathrm{P}(+|c,m) + \sum_{(c,m)\in \mathrm{B}_-} \log\left(1 - \mathrm{P}(+|c,m)\right)\right) \\
&= -\left(\sum_{(c,m)\in \mathrm{B}_+} \log \sigma(c\cdot m) + \sum_{(c,m)\in \mathrm{B}_-} \log\left(1 - \sigma(c\cdot m)\right)\right) \\
&= -\left(\sum_{(c,m)\in \mathrm{B}_+} \log \sigma(c\cdot m) + \sum_{(c,m)\in \mathrm{B}_-} \log \sigma(-c\cdot m)\right) \qquad (22)
\end{aligned}
$$

B_+ and B_- correspond to the positive and negative examples of the mini batch B respectively

To determine which samples to pick, we can sample based on the α-weighted unigram frequency of w_i, i.e.,

$$
\mathrm{P}_\alpha(w_i) = \frac{c(w_i)^\alpha}{\sum_{w'\in \mathrm{V}} c(w')^\alpha}
$$

The presence of α helps to smooth down the probability of words with high frequency, so that words with lower frequency will have a higher probability of being picked. Typically, α is set to 0.75.

6.3.2 Practical Considerations

In practice, the decisions affecting Word2Vec include:

- Data preprocessing steps, e.g.:
  - choice of tokenizer
  - case-folding
  - stemming/lemmatization
  - stopword removal
  - cross-sentence contexts
- Choice of hyperparameters, e.g.:
  - window size m

Additionally, Word2Vec has several limitations, including:

- Inability to represent multi-word phrases as words (e.g., ice cream).
- Inability to handle polysemy (i.e., words with multiple meanings) and words which can be used in multiple parts of POS.

- Inability to capture all semantics (e.g., antonyms) → limitation of the *distributional hypothesis*.
- Highly dependent on the training dataset.

6.4 *Properties of Embeddings*

On a high level, vector differences between embeddings yield semantic relationships (e.g., $\tilde{v}(\text{king}) - \tilde{v}(\text{man}) + \tilde{v}(\text{woman}) \approx \tilde{v}(\text{queen})$).

We can also create crosslingual embeddings by training a joint optimization over 3 terms:

- a source language,
- a target language, and
- a *regularization term* describing how difficult it is to linearly map between the two.

7 SEQUENCES

Lecture 7
4th March 2022

There are several types of tasks involving sequences, including:

- sequence classification (N → 1):
  - e.g., sentiment analysis, sentence factual checking.
- sequence labelling (N → N):
  - e.g., named-entity recognition, POS tagging.
- sequence to sequence (N → M):
  - e.g., machine translation, question answering, sentence simplification.
- sequence generation (1 → N):
  - e.g., image captioning.

7.1 *Parts of Speech*

Part-of-speech (POS), also known as word classes or syntactic categories, consists of several classes.

- **Nouns**: name of a person, place, thing, or idea.
- **Pronouns**: e.g., I, you, it, we, them, those.

- **Adjectives**: describes, modifies, or gives more information about a noun or person.
- **Verbs**: shows an action or a state of being.
- **Adverbs**: modifies a verb, an adjective, or another adverb. It tells how (often), where, and when.
- **Prepositions**: shows the relationship of a noun of pronoun to another word.
- **Conjunctions**: joins two words, ideas, or phrases together and shows how they are connected.
- **Interjections**: words or phrases which express a strong emotion, e.g., ouch!, hey!, oh!

Every word in the (English) vocabulary belongs to at least one of these classes.

There are two broad categories of POS, namely:

- **closed class**: small, fixed membership, typically consisting of *function* words (e.g., prepositions, pronouns, particles, determiners, etc.)
- **open class**: new vocabulary items can be created and added to them, e.g., nouns, verbs, adjectives, and adverbs.

7.1.1 POS Tagging

POS tagging describes the process of assigning a part of speech to words in a text. Specifically,

- the **input** consists of a sequence of observed tokenized words and a tagset, whereas
- the **output** consists of a sequence of unobserved (i.e., latent) tags.

POS tagging is useful as a first step for many tasks, e.g., named entity recognition, information extraction, and speech synthesis/recognition. However, words may be ambiguous occasionally, thus tagging can be seen as a disambiguation task (e.g., using supervised or unsupervised learning).

7.2 *Hidden Markov Models (HMM)*

Hidden Markov Models (HMM) are models based on the concept of Markov chains, and embodies the Markov assumption (i.e., memory-less property of a stochastic process). The components of a HMM include:

- $\mathrm{Q} = q_1 q_2 \ldots q_\mathrm{N}$: a set of **(latent) states**.
- $\mathrm{A} = a_{11} \ldots a_{ij} \ldots a_\mathrm{NN}$: a **transition probability** matrix.
  - each $a_{ij} = \mathrm{P}(q_j|q_i) = \frac{c(q_i q_j)}{c(q_i)}$ represents the probability of moving from state i to state j.
  - $\forall i, \sum_{j=1}^{\mathrm{N}} a_{ij} = 1$
- $\mathrm{O} = o_1 o_2 \ldots o_\mathrm{T}$: a sequence of **observations**.
  - each o_t represents the observation drawn from each time-step t from vocabulary $\mathrm{V} = v_1, v_2, \ldots, v_{|\mathrm{V}|}$.
- $\mathrm{B} = b_i(o_t)$: a sequence of **observation likelihoods** or **emission probabilities**.
  - each $b_i(o_t) = \mathrm{P}(o_t|q_i) = \frac{c(o_t, q_i)}{c(q_i)}$ expresses the probability of observation o_t generated from a state i.
- $\Pi = \pi_1, \pi_2, \ldots, \pi_\mathrm{N}$: an initial probability distribution over states.
  - each $\pi_i = \mathrm{P}(q_i| < s >) = \frac{c(<s> q_i)}{c(<s>)}$ represents the probability for the Markov chain to start at state i.
  - $\sum_{i=1}^{\mathrm{N}} \pi_i = 1$

7.2.1 Forward Computation

Given a HMM and a sentence with length T, forward computation can be invoked to compute the probability that the sentence has a label ordering $\mathrm{Q} = q_1 q_2 \ldots q_\mathrm{T}$. Specifically,

$$\mathrm{P}(\mathrm{O}, \mathrm{Q}) = \mathrm{P}(\mathrm{O}|\mathrm{Q}) \times \mathrm{P}(\mathrm{Q}) = \prod_{i=1}^{\mathrm{T}} \Big(\mathrm{P}(o_i|q_i) \times \mathrm{P}(q_i|q_{i-1}) \Big) \tag{23}$$

7.2.2 Viterbi Decoding

In contrast to naïve forward computation, Viterbi decoding helps us to determine the most probable sequence of labels for a given sentence, i.e.,

$$\mathrm{Q} = \arg\max_{q_1 \ldots q_\mathrm{T}} \prod_{i=1}^{\mathrm{T}} \Big(\mathrm{P}(o_i|q_i) \times \mathrm{P}(q_i|q_{i-1}) \Big)$$

Since we are only interested in knowing the *most probable* sequence of labels, we can employ dynamic programming and simply save previous states which

yield the highest probability. In other words, we have

$$
\begin{aligned}
v_t(j) &= \max_{i=1\ldots N} v_{t-1}(i) \times \mathrm{P}(q_j|q_i) \times \mathrm{P}(o_t|q_j) \\
&= \begin{cases} \pi_j b_j(o_1) & \text{if } t = 1 \\ \max\limits_{i=1\ldots N} v_{t-1}(i) a_{ij} b_j(o_t) & \text{if } t > 1 \end{cases}
\end{aligned} \tag{24}
$$

The time complexity of this algorithm is $O(TN^2)$, where T is the length of the sequence and N is the number of states in the HMM. A backtrace (indicated by *bt* in figure 13 below) can also be included in the algorithm to identify the sequence.

1. **Initialization:**

$$
\begin{aligned}
v_1(j) &= \pi_j b_j(o_1) \qquad 1 \le j \le N \\
bt_1(j) &= 0 \qquad 1 \le j \le N
\end{aligned}
$$

2. **Recursion**

$$
\begin{aligned}
v_t(j) &= \max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t); \quad 1 \le j \le N, 1 < t \le T \\
bt_t(j) &= \operatorname*{argmax}_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t); \quad 1 \le j \le N, 1 < t \le T
\end{aligned}
$$

3. **Termination:**

$$
\begin{aligned}
\text{The best score:} \quad P* &= \max_{i=1}^{N} v_T(i) \\
\text{The start of backtrace:} \quad q_T* &= \operatorname*{argmax}_{i=1}^{N} v_T(i)
\end{aligned}
$$

Figure 13: Formal definition for the Viterbi recursion.

8 ENCODER-DECODER MODELS

Lecture 8
11th March 2022

To model sequences well, we need to:

- handle **variable-length** sequences.
- track **long-distance** dependencies.
- maintain information about token **ordering**.
- **share parameters** across the sequence.

To address these problems, **recurrent neural networks (RNN)** could be utilized.

8.1 *Recurrent Neural Networks (RNN)*

Recurrent neural networks involve the inclusion of serial dependencies in neural networks. This is typically done by adding an edge between a unit and a copy of itself at the next time step.

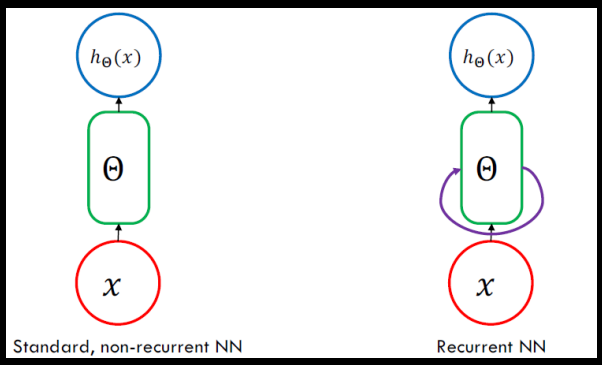

Figure 14: Simplified illustration of RNNs.

In other words, for subsequent time steps after the first one, the values for each (hidden) state h is dependent on both the input x and the value of the state in the previous time step:

$$h' = g_\theta(h, x') = g(\theta_{hh}h + \theta_{xh}x')$$

$$\hat{y}' = \theta_{hy}h'$$

where:

- θ_{hh} represent the set of *recurrent weights*,
- θ_{xh} represent the set of *input weights*, and
- θ_{hy} represent the set of *output weights*.

Once we have done a forward pass to generate predictions, we could utilize *backpropagation* to inform each weights on the extent of tuning needed to reduce the overall loss.

8.1.1 Conditional Language Models

A **conditional language model** assigns probabilities to sequences of words $w = w_1 w_2 \dots w_m$, given some conditioning context x. We can apply the chain rule to decompose this probability, i.e.,

$$\mathrm{P}(w|x) = \prod_{t=1}^{\mathrm{T}} \mathrm{P}(w_t|x, w_1, \dots, w_{t-1})$$

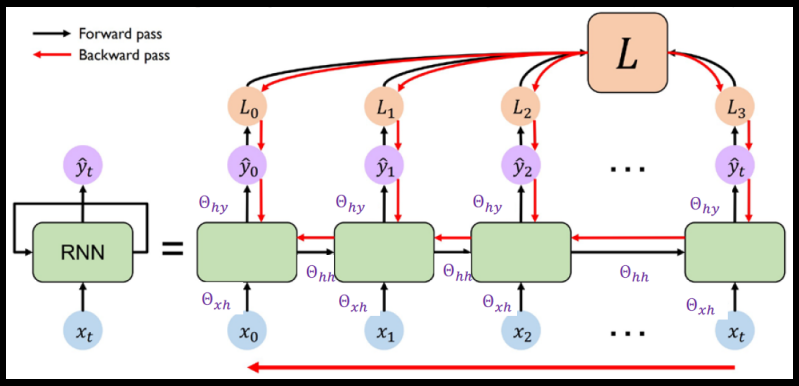

Figure 15: Illustration of loss backpropagated through time (BPTT).

8.2 *Encoder-Decoder*

Encoder-decoder models are models which learn a function that maps x into a fixed-size vector representation $\mathcal{R}$, and then uses a language model to "decode" that vector into a sequence of output words w. The basic 2-component setup involves:

- **Encoder**:
  - Learns a function that maps the context into a fixed-size vector representation $\mathcal{R}$.
  - Architecture depends on context (e.g., CNN for images, RNN for text).
- **Decoder**:
  - Language modelling using $\mathcal{R}$ to output a sequence of words.

We will look at some possible approaches for performing the mapping and incorporating the encoded context.

- **Kalchbrennen & Blunsom (2013)**:
  - Encoder:

$$c = \text{csm}(x)$$

$$\mathcal{R} = \theta_{cs}(c)$$

    a **convolutional sentence model (csm)** is used to map a sentence into a vector.
  - RNN decoder:

$$h_t = g(\theta_{hh} h_{t-1} + \theta_{xh} x_t + \mathcal{R}$$

$$w_t = \text{softmax}(\theta_{hy} h_t)$$

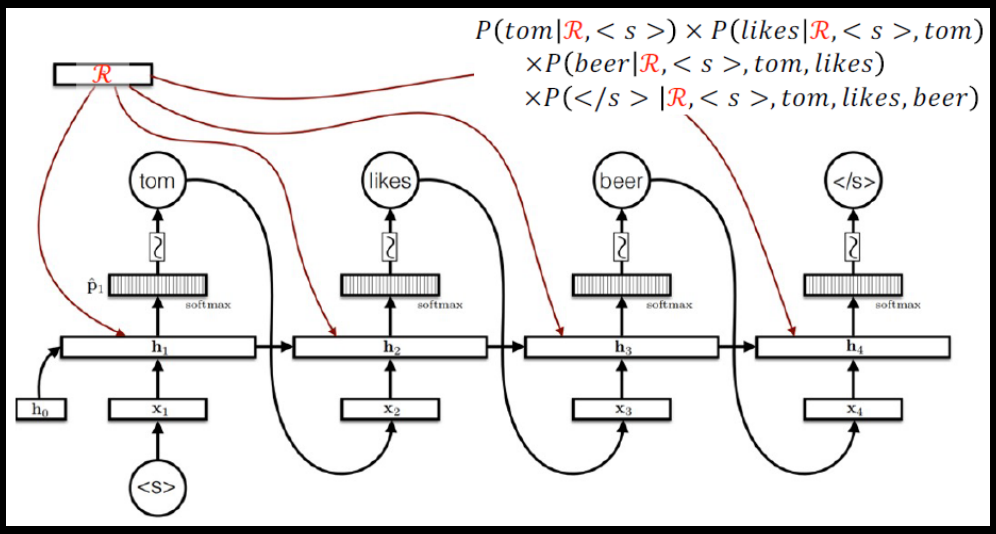

Figure 16: Sample visualization of the RNN decoder by K&B, 2013.

- **Sutskever et al. (2014)**:
  - RNN encoder:

$$h_t^{\text{enc}} = g(\theta_{hh}^{\text{enc}} h_{t-1}^{\text{enc}} + \theta_{xh}^{\text{enc}} x_t)$$

  - RNN decoder:

$$h_t^{\text{dec}} = g(\theta_{hh}^{\text{dec}} h_{t-1}^{\text{dec}} + \theta_{xh}^{\text{dec}} x_t)$$

$$\hat{w}_t = \text{softmax}(\theta_{hy}^{\text{dec}} h_t^{\text{dec}})$$

where $\mathcal{R} \equiv h_0^{dec} \equiv h_t^{\text{enc}}$.

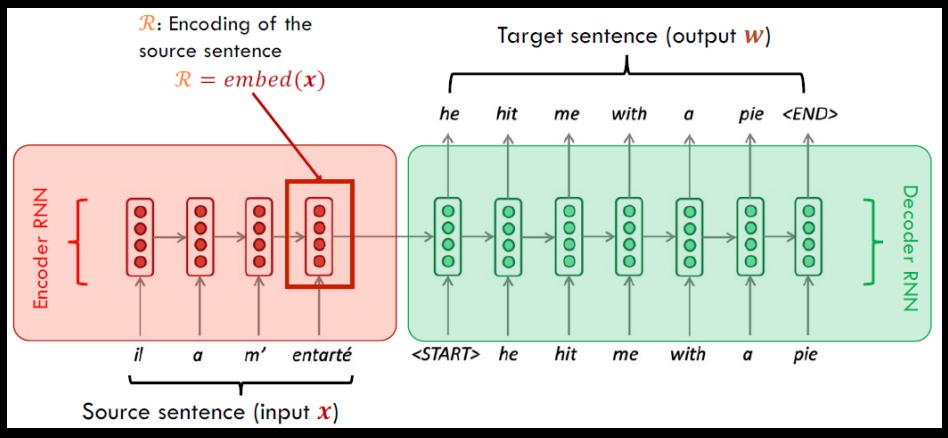

Figure 17: Sample visualization of the RNN decoder by Sutskever et al., 2014. Note that the context representation $\mathcal{R}$ is the final representation of the encoder RNN, and is used as the input to the decoder RNN.

8.3 *Attention*

the weighted sum is a *selective summary* of the information contained in the values, where the query determines which values to focus on

Given a set of vector values and a vector query, **attention** is a technique to compute a weighted sum of the values, dependent on the query. Attention provides several benefits, including:

- significantly improves *performance* by allowing the decoder to focus on certain parts of the source.
- solves the bottleneck problem (i.e., where the context representation $\mathcal{R}$ needs to be a fixed-size vector) by allowing the decoder to look directly at the source (and thus providing a way to obtain a *fixed-size representation of an arbitrary set of representations* dependent on some other representation).
- helps with the *vanishing gradient problem* by providing a shortcut to faraway states.
- provide some *interpretability* (e.g., by inspecting the attention distribution, we can see what the decoder is focusing on).

The core idea is to use a *direct connection to the encoder* on each step of the decoder, to *focus on a particular part of the source sequence*.

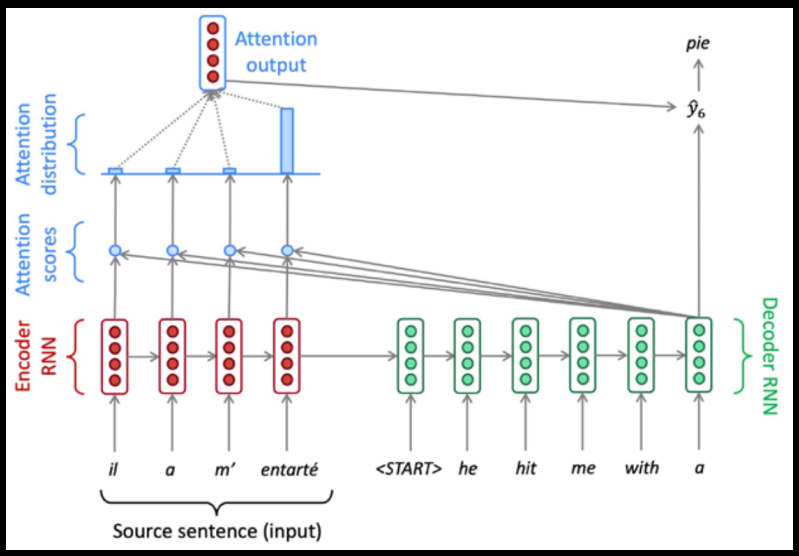

Figure 18: Illustration of attention in neural networks.

The implementation of attention involves the following steps.

1. There are encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$, and at each time step t, we have a decoder hidden state $d_t \in \mathbb{R}^h$.
2. We can retrieve the **attention score** e^t for this step:

$$e^t = [d_t^T h_1, \ldots, d_t^T h_n] \in \mathbb{R}^N$$

3. We then take a softmax to get the **attention distribution** a^t for this step (i.e., a probability distribution that sums to 1).:

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

4. We use a^t to make the **context vector (attention output)** a^t as a weighted sum of the encoder hidden states:

$$a^t = \sum_{i=1}^{\text{N}} a_i^t h_i \in \mathbb{R}^h$$

5. Finally, we concatenate the attention output a^t with the decoder hidden state d_t and decode the output:

$$w_t = \text{softmax}(\theta_{hy}[a^t; h_t])$$

where w_t represents the output vector.

8.4 *Beam Search Decoding*

There are several ways to generate/decode the target output:

- **Greedy decoding**: taking an *argmax* on each step of the decoder.
  - We decode until the model produces an < END > token.
  - There is no way to undo decisions → this approach is unlikely to find the global optimum, even though we would arrive at the local optima for each time step.
- **Exhaustive search decoding**: compute all possible sequences y.
  - This means that on each step t of the decoder, we are tracking V^t possible partial translations → the time complexity of $\text{O}(\text{V}^{\text{T}})$ is intractable.
- **Beam search decoding**: at each step of the decoder, we keep track of the k most probable partial translations (i.e., **hypotheses**).
  - k is the **beam size**, usually around 5 – 10.
  - We will keep track of the top k hypotheses $(y_1, \ldots, y_t)$ on each step, based on their scores:

$$\begin{aligned}\text{score}(y_1, \ldots, y_t) &= \log \text{P}_{\text{LM}}(y_1, \ldots, y_t | x) \\ &= \sum_{i=1}^{t} \log \text{P}_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)\end{aligned}$$

  - We continue beam search until:

    - We **reach time step** T, where T is some predefined cutoff.
    - We **have at least** n **completed hypotheses**, where n is a predefined cutoff.
  - Beam search is not guaranteed to find an optimal solution, but it is much more efficient than exhaustive search.
- **Sampling-based decoding**:
  - **Pure sampling**: on each time step t, we **randomly sample** from the probability distribution P_t to obtain our next word.
  - **Top-n sampling**: on each time step t, we randomly sample from P_t, restricted to just the **top-n most probable words**.

In general, greedy and beam search methods are deterministic, with beam search generally having better quality/performance over greedy decoding. However, if the beam size is too high, there is a chance of returning high-probability but unsuitable output (e.g., generic/short sentences).

On the other hand, sampling methods are more probabilistic/allows for more diversity, and could be used for open-ended/creative generation.

9 CONSTITUENCY PARSING

Lecture 9
18th March 2022

We have discussed about regular expressions previously, but there are some limitations associated with them:

- Not all languages can be described using RegEx (e.g., recursive languages such as $\{0^n 1^n | n \geq 0\}$).
- Specifically, natural language is not a regular language because it allows for nested structures/center embeddings.

To make sense of how meaning is mapped to language structures, we would need to first define the concept of **constituency** (a.k.a. **phrase structure**): a group of words that behave as a single unit. Based on this definition, sentences can be described as a hierarchical structure of constituents.

How do we know whether a group of words forms a constituent? There are two main ways to go about doing this:

- Informally: the group of words "makes sense" on its own.
- Formally: constituency tests, including:
  - **topicalization**: only a constituent can be moved to different locations in a sentence.
  - **pro-form substitution**: only a constituent can be substituted with a pro-form like *it*, *that*, *them*, *then*, *there*, etc.

  - **fragment answers**: only a constituent can answer a question, while retaining the meaning of the original sentence.

9.1 *Context Free Grammars (CFGs)*

A **context free grammar** gives a formal way to define what meaningful constituents are, and exactly how a constituent is formed out of other constituents (or words). In other words, it defines how symbols in a language combine to form valid structures.

CFGs are more powerful than RegEx, since they are capable of expressing recursive structures. They are made up of two types of symbols:

- **non-terminal symbols**: (unobserved) symbols that can be replaced according to rules (e.g., parts of speech, phrase names, etc.).
- **terminal symbols**: observed output of a rule, which cannot be changed or replaced further (e.g., words/tokens).

More formally, we can define a CFG as a 4-tuple (N, Σ, R, S), consisting of:

Symbol	Formal representation	Examples
N	Finite set of non-terminal symbols	NP, VP, S
Σ	Finite alphabet of terminal symbols	the, dog, a
R	Set of production rules, each $A \rightarrow \beta, \beta \in (\Sigma, N)$	$S \rightarrow$ NP VP, Noun $\rightarrow$ dog
S	Start symbol	

Figure 19: Definition of CFG.

Even with the use of CFGs, there exists the problem of **structural ambiguity** (i.e., when a grammar is able to assign more than one parse to a sentence). There are two common types of structural ambiguity, namely:

- **attachment ambiguity**: a particular constituent can be attached to the parse tree at more than one place.
- **coordination ambiguity**: phrases conjoined by a conjunction (e.g., "and") can be interpreted in multiple ways.

The fact that there are many grammatically correct parses, but some of which are semantically unreasonable, is an issue. There are two approaches which could be taken to solve this:

1. **syntactic parsing**: extract all possible parses for a sentence.
2. **syntactic disambiguation**: score all parses and return the best parse.

In order to evaluate parses, we could represent a parse tree as a collection of tuples $\{(l_1, i_1, j_1), \ldots, (l_n, i_n, j_n)\}$, where:

- l_k is the non-terminal labelling the k^{th} phrase,
- i_k is the index of the first word in the k^{th} phrase, and
- j_k is the index of the last word in the k^{th} phrase.

Specifically, we could convert the gold-standard tree and the system-hypothesized tree into such representations, and then estimate the precision, recall, and F1 scores.

9.2 *Chomsky Normal Form (CNF)*

The **Chomsky normal form** puts some constraints on the grammar rules, while preserving the same language. The benefit of this is that if a grammar is in CNF, we could avoid issues with ambiguity during parsing. Additionally, CNF provides an upper bound for parsing complexity (link).

The formal definition of CNF is nearly identical to that of CFG, with some additional constraints on the production rules.

Symbol	Formal representation	Examples
N	Finite set of non-terminal symbols	NP, VP, S
Σ	Finite alphabet of terminal symbols	the, dog, a
R	Set of production rules, each $A \rightarrow \beta$, β = single terminal (from Σ) or two non-terminals (from N)	S $\rightarrow$ NP VP, Noun $\rightarrow$ dog
S	Start symbol	

Figure 20: Definition of CNF.

Any CFG can be converted into a *weakly equivalent* CNF using two basic transformation steps:

1. Recursive removal (i.e., merging) of unary rules (and empty rules).
   - e.g.: (Nominal $\rightarrow$ Noun) + (Noun $\rightarrow$ book|flight) $\Longrightarrow$ Nominal $\rightarrow$ book|flight
2. Dividing n-ary rules (i.e., rules with > 2 non-terminals on the R.H.S.) by introducing new non-terminals.
   - e.g.: S $\rightarrow$ Aux NP VP $\Longrightarrow$ (S $\rightarrow$ X VP) + (X $\rightarrow$ Aux NP)

9.3 *Syntactic Parsing*

One means of extracting a sentential parse tree is by repeatedly parsing the sentence from left to right (i.e., a greedy approach). However, this greedy approach is unlikely to lead us to the gold-standard parse tree.

A better approach is the **CYK parsing algorithm**. The underlying idea is that given a context-free grammar G in CNF, we say that G can generate a constituent A comprising n words if there exists a rule $(A \rightarrow X\ Y) \in G$ where:

- X can generate $w_1 \dots w_k$, and
- Y can generate $w_{k+1} \dots w_n$.

This algorithm uses a bottom-up dynamic programming approach to handle redundancies/repeated computations. However, it requires the CFG to be in CNF, i.e.:

- the grammar is **epsilon transition free/ϵ-free** (i.e., could change state for free)
- either 2 non-terminal symbols or 1 terminal symbol on the R.H.S.

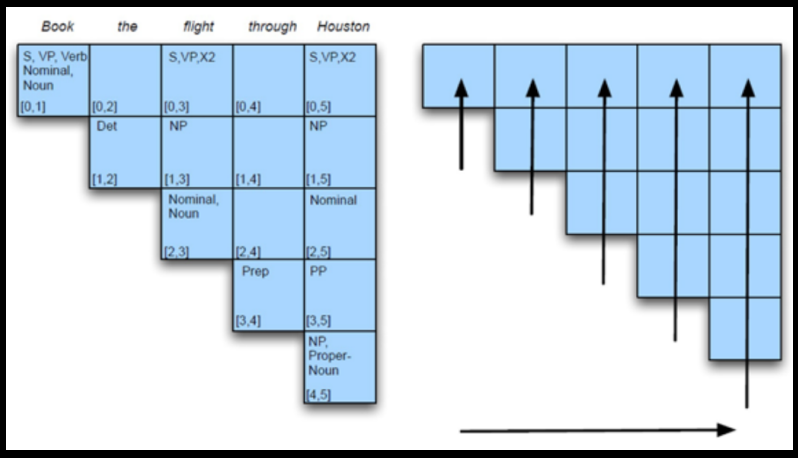

Figure 21: Illustration of the CYK parsing algorithm. Cells are filled from the bottom right to the upper right, in a left-to-right manner.

Specifically, each cell $c_{i,j}$ contains all the possible parses for the sub-sentence from word i to word j, and cell $c_{0,n}$ contains all the possible parses for a given input.

However, the basic CYK algorithm only solves the membership problem (i.e., the algorithm only checks if a sentence is a "member" of the language described by a grammar). Our goal is to get the most plausible parse tree for a sentence.

9.4 *Statistical Parsing*

To achieve our goal, we can apply **statistical parsing** instead, which helps to resolve structural ambiguity by choosing the **most probable parse**.

The application of statistical parsing requires the idea of a **probabilistic context-free grammar (PCFG)**, where each production is associated with a probability. More formally, a PCFG could be defined as a 4-tuple as follows:

Symbol	Formal representation	Examples
N	Finite set of non-terminal symbols	NP, VP, S
Σ	Finite alphabet of terminal symbols	the, dog, a
R	Set of production rules, each $A \rightarrow \beta[p]$, $p = P(\beta\|A)$	S $\rightarrow$ NP VP, Noun $\rightarrow$ dog
S	Start symbol	

Figure 22: Definition of PCFG.

Therefore, for a given parse tree T for sentence S comprised of n rules from R (each $A \rightarrow \beta$), we have

$$P(T, S) = \prod_{i} P(\beta|A)$$

Note that the probabilities with the same L.H.S. rule (e.g., NP $\rightarrow$ X, NP $\rightarrow$ Y, etc.) must sum to 1, i.e., $\sum_{\beta} P(\beta|A) = 1$, where each $P(\beta|A)$ can be obtained via counts from the labelled data:

$$P(\beta|A) = \frac{C(A \rightarrow \beta)}{\sum_{\gamma} C(A \rightarrow \gamma)}$$

While a CFG tells us whether a sentence is in the language it defines, a PCFG assigns scores to different parses for the same sentence.

To systematically extract from a sentential parse tree, we could adopt the probabilistic CYK algorithm, which is modified from the standard CYK algorithm with the following differences:

Standard CYK	Probabilistic CYK
Table$[i, j]$ = all possible parses for span $[i, j)$	Table$[i, j, A]$ = highest score for span $[i, j)$ resulting in constituent A
Table$[0, n]$ = all possible parses for the given input sentence	Table$[0, n, S]$ = highest score for a parse for the given input sentence

10 SEMANTICS

Lecture 10
25th March 2022

Syntax is the study of sentence structure (i.e., word ordering), whereas **semantics** is the study of word (and sentence) meaning.

Generally, syntax encodes the structure of a language, but syntax alone is not sufficient to ensure that a sentence is meaningful. Specifically, syntax does not **ground** a word. There are two main types of grounding:

ground: establishing the time, location, or actuality of a situation according to some reference point

- **Symbolic grounding**: connecting symbols (e.g., words) to perceptions of the real world.
  - e.g., seeing the symbol "apple" elicits a mental image of it, along with some of its characteristics.
- **Communicative grounding**: adding mutual knowledge, beliefs, and assumptions.
  - e.g., providing details of *who* is performing an action.

Additionally, syntax does not inform us how these referents change as a function of their compositionality.

Instead, we will have to consider **semantic representation**, i.e., formal structures composed from symbols to represent the meaning of sentences. We will be dealing with two types of semantics, namely:

- **Lexical semantics**: representing the meaning of *words*.
- **Logical semantics**: representing the meaning of *sentences*.

10.1 *Computational Lexical Semantics*

Computational lexical semantics refers to any computational process involving **word meaning**, e.g., computing word similarity, word relations, word connotation and sentiment, word sense disambiguation, etc. This comprises of the assessment of synonymy (i.e., how similar are two words to each other) and word relations (i.e., part-whole or supertype-subtype).

There are several important terminology which will be used to discuss word relations:

- **Lemma/Stem**: canonical/base form of an inflected word.
- **Lexeme**: set of all the inflected forms of a lemma.
- **Sense**: meaning of a word.
- **Polysemy**: words/phrases with multiple sense.

  - We can assess whether a word has more than one sense using the *zeugma test*.
- **Homonymy**: similar words with different sense/meaning.
  - **Homophone**: different spelling, same sound.
  - **Homonym**: same spelling, same sound.
  - **Homograph**: same spelling, different sound.
- **Metonymy**: a systematic relationship between senses.
- **Synonymy**: words/phrases with the same meaning in some or all contexts.
  - Synonymy could be weak (e.g., only similar in few contexts) or strong/perfect (e.g., semantically identical).
  - However, synonyms may not preserve acceptability based on *pragmatics* (e.g., notions of politeness, slang, genre, etc.).
- **Antonymy**: words/phrases with opposite meanings in certain contexts. Antonymy can be further elaborated into different types, including:
  - **Gradable antonymy**: on a scale, e.g., hot/cold, big/small.
  - **Complementary antonymy**: property of a single entity, e.g., married/single, alive/dead.
  - **Converse antonymy**: pairs of opposites where one cannot exist without the other, e.g., husband/wife, above/below, buy/sell.
- **Hyponymy**: one sense is a hyponym of another if the first sense is more *specific*, denoting a subclass of the other.
  - More formally, a sense A is a hyponym of a sense B if being an A *entails* being a B.
  - Hyponymy is usually transitive.
  - Also known as the **IS-A hierarchy**.
- **Hypernymy**: a superclass/superordinate relationship.
- **Meronymy**: the "part" in a part-whole relationship (e.g., wheel is a meronym of car).
- **Holonymy**: the "whole" in a part-whole relationship (e.g., car is a holonym of wheel).

10.1.1 WordNet

WordNet is a *hierarchically organized* lexical database, consisting of:

- **Synsets** (i.e., synonym sets): group of word sense that are (near-)synonyms.
  - Each synset comes with a **gloss** (i.e., a short description/definition).

- Noun relations: e.g., hypernymy, hyponymy, meronymy, holonymy, and synonymy/antonymy.
- Verb relations: e.g., hypernymy, troponymy, entailment, causation, pertainymy, and synonymy/antonymy.

troponymy: from events to a subordinate event, e.g., walk → stroll

pertainymy: e.g., vocal pertains to voice

10.1.2 Word Similarity

There are two main classes of similarity algorithms:

- **Distributional algorithms**: algorithms which make use of the distributional hypothesis (e.g., Word2Vec).
- **Thesaurus-based algorithms**: are words "nearby" in a hierarchy? do they have similar glosses?

gloss: brief meaning of a word

We can use some of the following approaches to determine whether two words are nearby:

- **Path-based similarity**: two concepts are similar if they are near each other in the thesaurus hierarchy. Specifically, we define

$$\text{pathLength}(c_1, c_2) = 1 + \text{numEdgesBetween}(c_1, c_2)$$

$$\text{simPath}(c_1, c_2) = \frac{1}{\text{pathLength}(c_1, c_2)}$$

  - However, basic path similarity assumes that each link represents a uniform distance (which typically isn't the case).
- **Resnik similarity**: the information content of the most informative subsumer (i.e., parent/ancestor node) of the two nodes. Specifically, we define:
  - words(c) be the set of all words that are children of node c
  - P(c) as the probability that a randomly selected word in a corpus is an instance of concept c, i.e.:

$$\text{P}(c) = \frac{\sum_{w \in \text{words}(c)} \text{count}(w)}{\text{N}}$$

  - the information content of a node as

$$\text{IC}(c) = -\log \text{P}(c)$$

  - the **lowest common subsumer** $\text{LCS}(c_1, c_2)$ = the most informative (i.e., lowest) node in the hierarchy subsuming both c_1 and c_2.

  Thus,

$$\text{sim}_{\text{resnik}}(c_1, c_2) = -\log \text{P}\Big(\text{LCS}(c_1, c_2)\Big) \tag{25}$$

On the other hand, we can also say that two concepts are similar if their glosses *contain similar words*.

- For each n-gram phrase that is contained in both glosses, we would add a score of n^2 to the similarity score.

10.2 *Logical Semantics*

Ideally, our meaning representation language should have the following properties:

1. **Verifiability**: ability to match the meaning representation of a sentence against representations of facts about the world modelled in the system's knowledge base.
    - Yes: representation matches an existing fact
    - No: representation *contradicts* an existing fact
    - Don't know: knowledge base is incomplete
2. **Unambiguous representations**: each statement in a meaning representation should have only one meaning.
3. **Canonical form**: different sentences with the same meaning should be assigned the same meaning representation.
4. **Inference**: we should be able to draw valid inferences not explicitly stated in the meaning representation of the input sentence/program's knowledge base.
5. **Expressiveness**: the meaning representation language must be able to adequately express and represent (i.e., a wide range of language concepts can be represented) the meaning of natural language sentences.

10.2.1 First Order Logic

To satisfy the above properties, we could represent every sentence as an unambiguous proposition in **first-order logic (FOL)**, which consists of the following components:

- **Constants**: specific entities in the world being described.
- **Relations**: relationships that hold among FOL terms (i.e., constants, variables, functions).

  - **Properties** denote sets of elements in the domain, and are *unary relations* (e.g., [[Human]] = {Alice, Bob, Trudy}).
  - (n-ary) Relations denote sets of tuples in the domain (e.g., [[Likes]] = $\{< Alice, Bob >, < Alice, Trudy >\}$).

  Specifically, a relation takes terms as arguments, and results in a sentence that has a truth value.
- **Functions**: take terms as arguments, and result in another term, denoting an object.
  - An **extensional definition** of a concept/term formulates its meaning by specifying its extension (i.e., every object that falls under the definition of the concept in question).
- **Variables**: non-committed placeholders for terms (e.g., Likes(Alice, x)).
- **Quantifiers**: expressions to assign a truth value to a variable (e.g., $\exists x$, Likes(Alice, x)).
- **Logical connectives**: formal means for composing expressions in a meaning representation language (e.g., symbols, quantifiers). Some sample boolean operations on connectives include:
  - Negation ($\neg\phi$)
  - Conjunction ($\phi \wedge \psi$) and disjunction ($\phi \vee \psi$)
  - Implication ($\phi \implies \psi$)
  - Equivalence ($\phi \Leftrightarrow \psi$)

The propositions would adhere to model-theoretic semantics, i.e., the truth of a proposition is determined w.r.t. some model $\mathcal{M}$ of the world. We will use $[[\cdot]]_{\mathcal{M}}$ to describe a **denotation** (i.e., the literal meaning of the term) of a term in a specific model $\mathcal{M}$.

FOL lets us evaluate the truth conditions about specific entities and relations in $\mathcal{M}$.

10.2.2 Lambda Calculus

The **principle of compositionality** states that the meaning of a complex expression is a function of the meaning of its constituent parts/syntactic constituents (e.g., meaning(S) = function(meaning(NP), meaning(VP))). To apply the principle of compositionality to FOL, we could utilize **lambda calculus**.

lambda calculus: functions that can be applied to FOL terms as arguments to yield new FOL expressions, where the variables are bound to the argument terms

Specifically, we could use λ-**reduction**, where lambda expressions and repeatedly unified with terms. For example:

- Expression: $\lambda y \lambda x \text{Likes}(x, y)$, terms: Alice, Bob

1. $\lambda y \lambda x \text{Likes}(x, y)(\text{Alice}) \longrightarrow \lambda x \text{Likes}(x, \text{Alice})$
2. $\lambda x \text{Likes}(x, \text{Alice})(\text{Bob}) \longrightarrow \text{Likes}(\text{Bob}, \text{Alice})$

In general, we can let the non-terminals determine how the semantics of their constituents are combined.

10.2.3 Adjuncts and Roles

When dealing with categories (e.g., human), instead of creating a unary predicate for each category (e.g., human(Alice)), we could apply **reification**, i.e., represent all concepts that we want to make statements about as full-fledged objects (e.g., ISA(Alice, human)).

This is somewhat similar to abstraction in OOP

Similarly, when dealing with adjuncts (i.e., optional parts of a statement), instead of extending the arity of the relation (e.g., Likes(Alice, Bob) $\longrightarrow$ Likes(Alice, Bob, 8 out of 10, unrequited)), we can reify the event to an existentially quantified variable of its own, and then use it as an argument in other relations. For example:

$$\exists x, \text{Person}(x, \text{Bob}) \wedge \text{Likes}(\textbf{Alice}, x) \wedge \text{Score}(x, 8) \wedge \text{Status}(x, \text{unrequited})$$

11 CLASSIFICATION APPLICATIONS

Lecture 11
1st April 2022

11.1 *Summarization*

Text summarization involves the producing of an *abridged version* of a text that contains information which is important or relevant to a user. Some of the key dimensions in summarization tasks include:

1. What to summarize?
   - Single-document summarization: e.g., produce the abstract, outline, and headline of a document.
   - Multi-document summarization: e.g., produce the news stories on the same event given a group of documents.
2. Generic or query-focused?
   - Generic summarization: summarize the content of a document generally.

- Query-focused summarization: summarize a document w.r.t. an information need expressed in a user query (i.e., information retrieval). Some example use cases include:
  - Creating snippets summarizing a webpage for a query.
  - Creating a single cohesive *answer passage* answering complex questions, by combining information across multiple documents.

3. Extractive or abstractive?
   - Extractive summarization: creating summaries from phrases or sentences in the source document(s).
   - Abstractive summarization: expressing the ideas in the source documents using different words.

Extractive summarization could be considered both discriminative and generative, whereas abstractive summarization is mostly generative.

There are 3 main steps in a basic summarization algorithm:

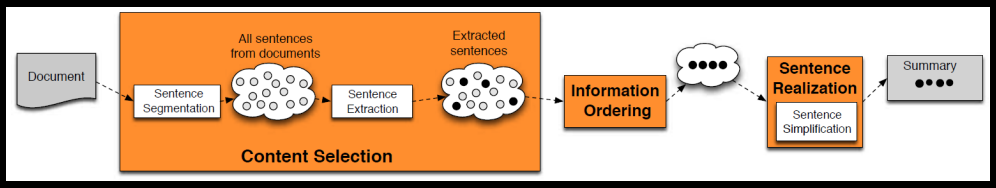

Figure 23: Steps in a basic summarization algorithm.

1. Content selection: choosing the key information.
   - One possible approach involves selecting sentences which have salient or informative words.
   - For example, we could weigh each word w_i in document d_j by its tfidf; subsequently, we would choose a smaller set of salient words by its cosine similarity with the query (cf. CS3245).
   - An alternative approach is TextRank.
2. Information ordering: choosing the order for placing content.
3. Sentence realization: cleaning up the text.

11.1.1 Summary Evaluation

One metric used in evaluating summaries is **ROUGE (Recall Oriented Understudy for Gisting Evaluation)**, based on the concept of n-gram overlap.

Given a document d, a set of reference summaries $\mathrm{Y} = \{y_1, \ldots, y_n\}$, and an automatic summary $\hat{y}$, ROUGE returns the percentage of n-grams from Y

which appear in $\hat{y}$. Specifically,

$$\text{ROUGE-}n = \frac{\sum_i \text{common\_}n\text{grams}(y_i, \hat{y})}{\sum_i \text{total\_}n\text{grams}(y_i)}$$

Even though ROUGE is not as good as human evaluation, it has been statistically proven to be a much more convenient proxy.

11.1.2 Query-Based Summarization

There are two main approaches to query-focused summarization, namely:

- (Bottom-up) snippet method:
  1. Find a set of relevant documents.
  2. Extract informative sentences from the documents.
  3. Order and modify the sentences into an answer.
- (Top-down) information extraction method:
  - Build specific answerers for different question types, e.g., definition/biography questions.

Some other techniques which could be employed in multi-document summarization include:

- Sentence simplification: parsing sentences, and then using learned rules to prune modifiers (e.g., attribution clauses, initial adverbials, etc.).
- Maximal marginal relevance (MMR): greedily pick the best sentence to add to the existing summary iteratively.
  - Specifically, we would want to pick sentences which are **relevant** (to the user's query) and **novel** (i.e., minimally redundant w.r.t. the existing summary).
  - The equation for selecting the best sentence based on MMR is given by

    $$s_{\text{MMR}} = \max_{s \in \text{D}} \left(\alpha \cdot \text{sim}(s, \text{Q}) - (1 - \alpha) \max_{s \in \text{S}} \left(\text{sim}(s, \text{S}) \right) \right)$$

    where D represents the set of documents, Q represents the query, and S represents the current summary. Cosine similarity could be employed to return the similarity scores between sentences.

After summarization, we would want to decide on how to order the sentences in our summary. Some possible alternatives for information ordering include:

- Chronological: ordering sentences by the date of the document.
- Coherence: choosing orderings where neighbouring sentences are similar/discuss the same entity.
- Topical: learning the ordering of topics in the source documents, and ordering sentences based on these topics.

We may also want to extract information particular to the query, e.g., a query w.r.t. the biography of a person would include his birth/death dates, education, nationality, etc.

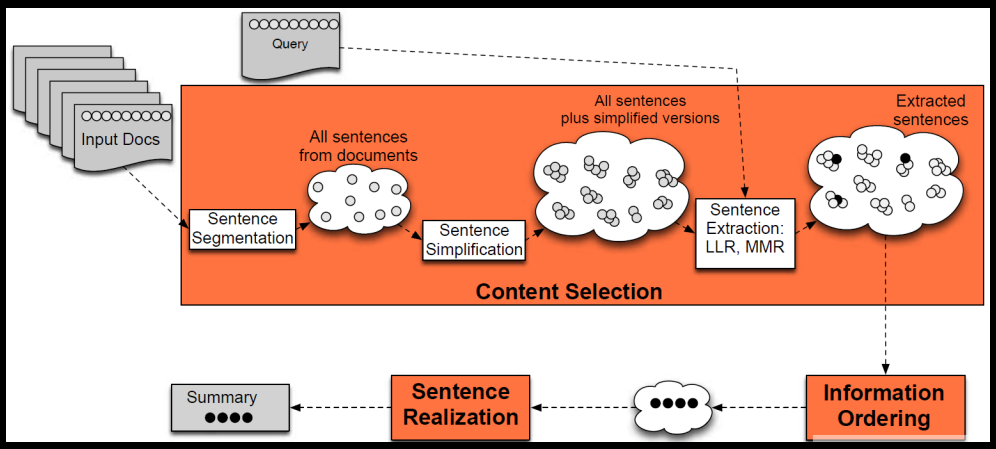

Figure 24: Sample architecture for query-based summarization.

11.2 *Question Answering*

Question answering (QA) systems typically involve the following components:

- Questions: e.g., factoid (i.e., "who did XXX?") or complex (i.e., "what does X think about YYY?").
- Answers: e.g., binary, short-answer, or MCQ/MRQ.
- Context: e.g., a passage/document, an image/video, or some knowledge base.

There are two main paradigms for QA, namely:

- IR-based approaches (e.g., Google). This are generally 3 main steps performed by these models:
  1. **Question processing**: detecting question and answer types, formulate queries to send to a search engine/database.
  2. **Passage retrieval**: retrieve ranked documents, then break into suitable passages and re-rank.

3. **Answer processing**: extract candidate answers, and rank candidates using evidence from the text and external sources.

- Knowledge based approaches (e.g., Siri, Wolfram Alpha): extracting knowledge from a structured knowledge base. This often involves:
  1. Building a semantic representation of the query.
  2. Mapping to a query structure which could be used to query structured data/resources (e.g., ontologies: WordNet, dbPedia, scientific databases, geospatial databases).
- Hybrid approaches (e.g., IBM Watson): useful for finding answers which are not yet injected into a structured knowledge base. This involves:
  1. Building a shallow semantic representation of the query.
  2. Generate answer candidates using IR methods.
  3. Scoring each candidate using richer knowledge sources (similar to in KB approaches).

11.2.1 IR-based QA Systems

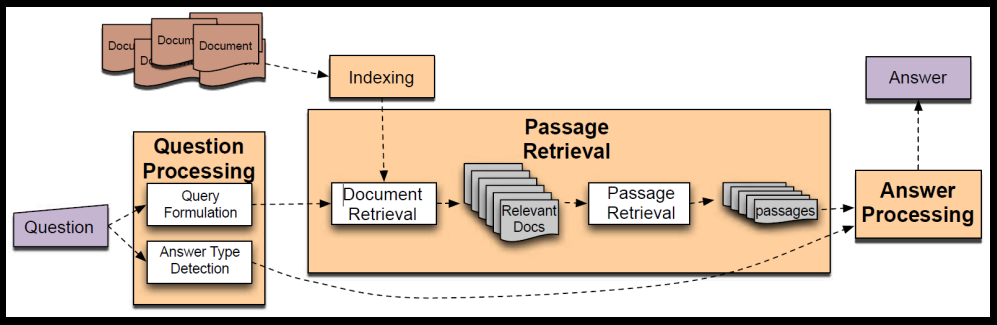

Figure 25: Sample architecture for IR-based factoid QA systems.

During **question processing**, we would need to extract some information from the question, which may include:

- answer type: e.g., the named entity type (e.g., person or place) of the answer.
  - This could be described either on the coarser level (e.g., "location", "entity") or on a finer level (e.g., "city", "state", "animal").
  - The detection of answer types could involve the use of regular expression-based rules, or the question headword (i.e., the first noun phrase after the *wh*-word).
  - This detection/classification could also be seen as a supervised ML task, involving the following steps:
    1. Define a taxonomy of question types.
    2. Annotate training data for each question type.

3. Train classifiers using a set of features (e.g., question words/phrases, POS tags, parse features/headwords, etc.).

- query formulation.
  - This could be done by selecting (ranked) keywords from the question for the query to be passed to the IR system.
  - The ranking of keywords could be done in several ways, such as that proposed by Moldovan et al. (1999), listed below:
    1. Select all non-stop words in quotations.
    2. Select all NNP (i.e., proper noun) words in recognized named entities.
    3. Select all complex nominals with their adjectival modifiers.
    4. Select all other complex nominals.
    5. Select all nouns with their adjectival modifiers.
    6. Select all other nouns.
    7. Select all verbs.
    8. Select all adverbs.
    9. Select the QFWs (i.e., question focus words).
    10. Select all other words.
- question type.
- focus: which are the words that are most important in finding the answer.
- relations (between entities in the question).

Subsequently, **passage retrieval** is performed, which involves the following steps:

1. Retrieval of documents using query terms.
2. Segmentation of documents into shorter units (e.g., paragraphs).
3. Passage ranking, e.g., by examining passages for:
   - Number of named entities of the right type.
   - Number of query words.
   - Number of question n-grams.
   - Proximity of query keywords.
   - Longest sequence of question words.
   - Rank of the document containing the passage.

Finally, **answer extraction** involves running an answer-type named-entity tagger on the passages, and returning strings with the right type (e.g., city). Features which could be taken into account when ranking candidate answers include:

- Answer type match: candidate contains a phrase with the correct answer type.

- Pattern match: regular expression pattern matches the candidate.
- Number of question keywords in the candidate.
- Keyword distance: distance in words between the candidate and query keywords.
- Novelty factor: a word in the candidate is not in the query.
- Apposition features: the candidate is an appositive to question terms.
- Punctuation location: the candidate is immediately followed by a punctuation.
- Sequences of question terms: the length of the longest sequence of question terms that occurs in the candidate answer.

11.2.2 Knowledge-Based QA Systems

Aside from IR approaches, we can also return answers to questions using knowledge bases, e.g., graphical knowledge bases such as Freebase. In graphical knowledge bases,

- Nodes represent entities.
- Edges represent relationships between entities.

For answer extraction, since we are using relations from databases, we can extract these relationships easily and convert them into FOL to be inserted into a knowledge base. Subsequently, we could employ tools covered in Section 10.2 to resolve the relationships.

Knowledge-based systems can also be used in temporal reasoning, as well as context and conversation in virtual assistants.

11.2.3 Evaluation Metrics for QA

Some common evaluation metrics for QA include:

- Accuracy
- Precision, recall, and F_1
- Mean Reciprocal Rank (MRR):
  - For each query, we return a ranked list of M candidate answers.
  - The query score is then $\frac{1}{\text{rank}}$ of the first correct answer.
  - We will take the mean over all n queries.

Lecture 12
8th April 2022

12.1 *Machine Translation*

To get *contextual* word embeddings, we can utilize context-independent, pre-trained language models (e.g., Word2Vec, or some language model trained on a corpus crawled from the internet) and make them context/task sensitive. The fine-tuning of word embeddings can be done by learning new weights for the pre-trained model, such that it generalizes better to the current task at hand (e.g., translation, summarization, etc.). This process of **transfer learning** is described in the diagram below:

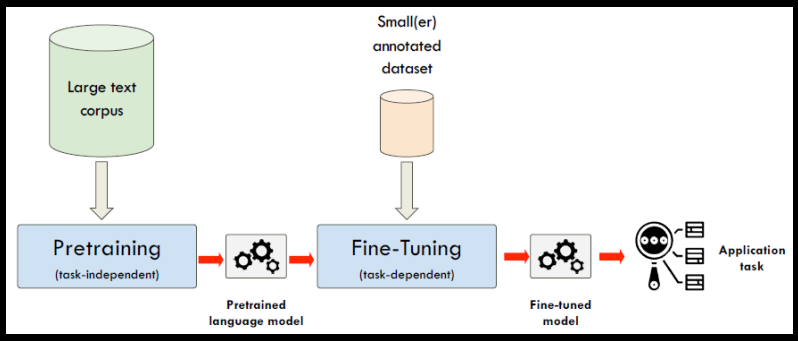

Figure 26: Transfer learning for NLP models.

While pretrained word embeddings attempt to predict words based on their co-occurrence, disregarding the surrounding context, pretrained contextualized embeddings (e.g., ELMo, BERT) attempt to predict words based on their surrounding context or the entire text.

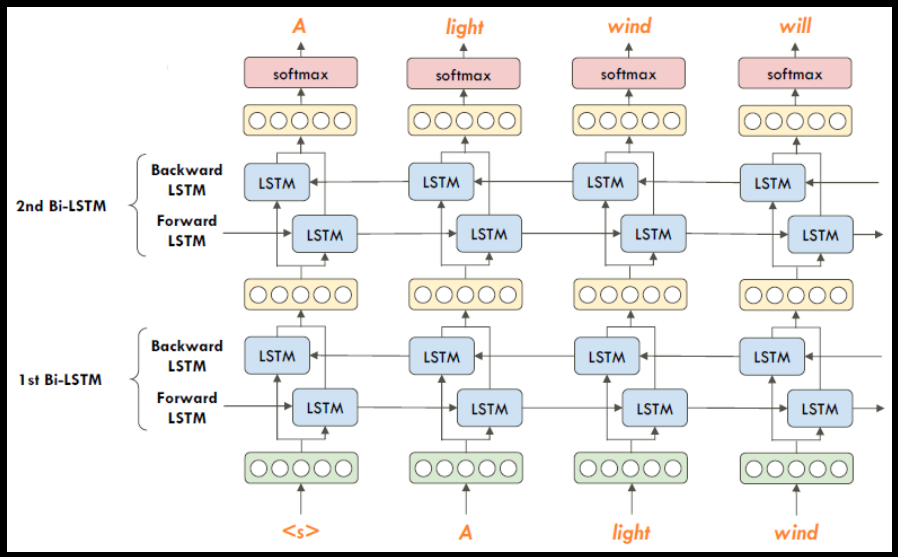

Figure 27: Illustration of ELMo.

For example, ELMo consists of 2 Bi-LSTM layers, with the first layer creating an intermediate representation from word-level tokens, and the second layer composing the intermediate representation to assemble the final version of the word embeddings. The final embeddings may be either taken directly from the output of the topmost layer, or a weighted sum of the final and intermediate contextual embeddings, i.e.:

$$\text{emb}_t = \gamma \sum_{j=0}^{k} s_j h_t^{(j)}$$

for a network with k layers, where:

- γ is a task-dependent scaling factor,
- s_j is a weighing factor for each layer, and
- $h_t^{(i)}$ is the contextualized embedding for the t^{th} token at the i^{th} layer.

Typically, the lower layer(s) is better for capturing lower-level syntax (e.g., POS, dependency parsing, NE recognition), whereas the higher layer(s) is better for capturing higher-level semantics.

12.2 *Noisy Channel Model*

Machine translation can also be seen as a code-breaking task, where the task of the language model is to learn to decode a message which has been passed through a noisy channel.

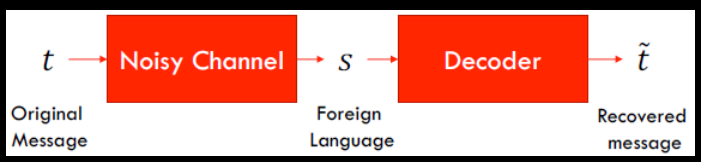

Figure 28: Machine translation as a code-breaking task.

This is akin to generative modelling using Bayes' rule:

$$\hat{t} = \arg\max_t \text{P}_{\text{LM}}(t) \times \text{P}_\theta(s|t)$$

On the other hand, machine translation can also be done using direct modelling/pattern matching (i.e., a discriminative approach). This approach is known as **neural machine translation (NMT)**, whereas the former approach is known as **statistical machine translation (SMT)**.

While NMT requires significantly more data than SMT to have desirable performance, modern machine translation systems are largely relying on

NMT due to their significantly better performance over SMT.

12.2.1 Evaluation Metrics for MT

While manual evaluation of MT is most accurate, it is too expensive. Automated evaluation metrics are often used to assess the performance of MT instead, including:

- comparing of system hypothesis with reference translations
- modified n-gram precision:

$$p_n = \frac{\#n\text{-grams appearing in both reference and hypothesis translations}}{\#n\text{-grams appearing in the hypothesis translation}}$$

- **Bilingual Evaluation Understudy (BLEU)**:

$$\text{BLEU} = \text{B} \times \exp\left(\frac{1}{\text{N}} \sum_{n=1}^{\text{N}} \log p_n\right)$$

  where $\text{B} = \exp\left(1 - \frac{r}{h}\right)$ is the **brevity penalty**, which penalizes translations which are shorter than the reference (i.e., r = reference length, h = hypothesis length).
  - To avoid logging 0, all precisions p_n can be smoothed.
  - Each n-gram in the reference can be used at most the number of times in the reference translation.

As compared to ROUGE (from section 11.1.1) which is recall-oriented, BLEU is precision-oriented. These two metrics can be combined to give an F_1 score:

$$\text{F}_1 = 2\frac{\text{BLEU} \cdot \text{ROUGE}}{\text{BLEU} + \text{ROUGE}}$$

* * *