Statistical Natural Language Processing

Prof. Dr. Axel Ngonga

Student lecture notes by
Tanja Tornede & Alexander Hetzer
Status: August 20, 2018

Contents

CHAPTER 1

Introduction

1.1 Origins

1.1.1 Linguistic Science

- **definition**: aim of a linguistic science is to be able to **characterize and explain** the multitude of linguistic observations circling around us
- three main concerns:
  - cognitive: how do humans acquire, produce and understand language?
  - model: how are linguistic utterances related to the real world?
  - structural: by which means to languages communicate semantics?

1.2 Goal of SNLP

- **definition**: find **common patterns** that occur in language use and exploit them to process natural language automatically
- two main groups of people
  - **rationalists**: natural language processing cannot be modeled by statistics
  - **empiricists**: statistical considerations are essential to an understanding of NL
    - processing of NL automatically demands finding set of regularities
    - clear that they do not always exist
    - ability to teach language systematically points to automatic processing
    - complex statistical models can predict and capture rare phenomena

⇒ we follow empiricists

1.3 Applications

- machine translation
- knowledge extraction (transform text into structured knowledge)
- knowledge graphs (fact checking, efficient queries, verbalization)
- question and answering
- sentiment analysis
- text summarization
- natural language generation

CHAPTER 2

Text Normalization

2.1 Eliza

- one of the first chatbots
- uses **pattern matching** and **substitutions** to simulate psychologist
- simplest implementation: regular expressions and text normalization

2.2 Regular Expressions

- formal language for specifying text strings
- allows to search for similar forms of a word (e.g. woodchuck, Woodchuck, woodchucks...)

2.2.1 Assumptions

- process one document at a time
- every document is a **sequence** of lines

2.2.2 Operators

Disjunction

- simple disjunctive patterns

Pattern	Matches
woodchuck	Fails to find Woodchuck
[wW]oodchuck	Woodchuck, woodchuck
[1234567890]	Any digit

- pipe as disjunction

Pattern	Matches
groundhog|woodchuck	My woodchuck is blue
yours|mine	This is yours.
a|b|c	= [abc]
[gG]roundhog|[Ww]oodchuck	Woodchucks aren't blue

- ranges

Pattern	Matches	
[A-Z]	An upper case letter	Drenched Blossoms
[a-z]	A lower case letter	my beans were impatient
[0-9]	A single digit	Chapter 1: Down the Rabbit Hole

- negations: Carat (^) means negation only when first in []

Pattern	Matches	
[^A-Z]	Not an upper case letter	Oyfn pripetchik
[^Ss]	Neither 'S' nor 's'	I have no exquisite reason"
[^e^]	Neither e nor ^	Look here
[a^b]	The pattern a carat b	Look up a^b now

Kleene Operators

- form of wildcards

Pattern	Matches	Example
colou?r	Optional previous char	color colour
oo*h!	0 or more of previous char	oh! ooh! oooh! ooooh!
o+h!	1 or more of previous char	oh! ooh! oooh! ooooh!
baa+		baa baaa baaaa baaaaa
beg.n		begin begun begun beg3n

Beginning and End of Line

Pattern	Matches
^[A-Z]	Palo Alto
^[^A-Za-z]	1 "Hello"
\\.$	The end.
.$	The end?The end!

More Operators

RE	Expansion	Match	Examples
\d	[0-9]	any digit	Party of 5
\D	[^0-9]	any non-digit	Blue moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[\r \t \n \f]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in Concord

2.2.3 Kinds of Errors

1) false positive (type 1): expression matches things which it should NOT match

2) false negatives (type 2): expressions does not match things which it should match

2.2.4 Substitutions

- operator: *s/<regexp>/<replacement>/*
- e.g.: *s/colour/color/*: replaces "colour" by "color"

Remembering Words to Replace

- sometimes we want to replace a word with a form of it where we added a part ⇒ remembering the string which matches is required
- **number operators/registers**: e.g.: *s/([0-9]+)/a\1a/*
  - adds "a" around numbers
- usage: use round brackets () to surround regex and *number* to refer to the word matching the regex
- can also be used for complex pattern matching
  - e.g.: "*the (.*) they were, the \1 they became*" matches "the hungrier they were, the hungrier they became"
- multiple occurrences of "()" can be referred to by multiple numbers (i.e. first bracket is referred to by 1, second by 2, etc.)

2.3 Finite State Automata

2.3.1 Formal Definition

FSA $(Q, \Gamma, q_0, F, \delta)$ where

- $Q = (q_0, q_1, \ldots, q_{N-1})$: finite set of N **states**
- Γ: finite **input alphabet** of symbols
- q_0: start state
- $F \subseteq Q$: set of **final states**
- $\delta : Q \times \Gamma \to Q$: transition function

2.3.2 Deterministic FSA

- FSA is **deterministic** if we have no ϵ transitions and a given input symbol clearly indicates which transition has to be used in all states

2.3.3 Deterministic Recognition

- given a formally specified DFSA, we can check if a given character sequence is part of a language or not

```
function D-RECOGNIZE(tape,machine) returns accept or reject
index ← Beginning of tape
current-state ← Initial state of machine
loop
    if End of input has been reached then
        if current-state is an accept state then return accept
        else return reject
    elsif transition-table[current-state,tape[index]] is empty then
        return reject
    else
        current-state ← transition-table[current-state,tape[index]]
        index ← index + 1
end
```

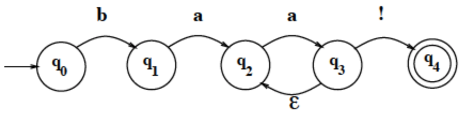
- can be adapted in order to find matches in text (i.e. instead of "reject", restart process from the second token on which we have considered)

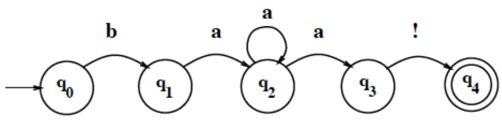
2.3.4 Formal Languages

- formal language definition: set of strings from Γ^*
- model m which can both **generate and recognize all and only** the strings of a formal language acts as a definition of the formal language
- $L(m)$: formal language characterized by m
- models are often useful as they are finite but express an infinite language

2.3.5 Non-Deterministic FSA

- FSA is called **non-deterministic**, if the behavior of the automaton in any state is NOT uniquely determined by its current state and the given input
- two main forms:
  - ϵ-transition: allows to move to another state without reading any input
  - two transitions with the same character leaving a state





2.3.6 Non-Deterministic Recognition

three main approaches

- backup-strategy: if taking a choice does not work out, we jump back to the latest choice and take another option
  - need to remember where our choice points are and which paths we already explored (i.e. search algorithm!)
- look-ahead: we look ahead in the input to help decide which option to take
- parallelism: we look at alternative paths in parallel

2.4 Formal Languages vs. Regular Expressions vs. FSAs

- any FSA can be described by a regular expression
- any regular expression (except one which make use of memory registers) can be implemented as FSAs
- both regular expressions and FSAs can describe a **regular language**

2.4.1 DFSAs vs. NFSAs

- NFSAs and DFSAs have the same expression power
- for any NFSA there exists a formally equivalent DFSA
  - can be constructed via the power method
  - idea: construct equivalence state for any states which can be reached via the same symbol from a state
  - if NFSA has N states, constructed DFSA might have up to 2^N states

2.5 Text Normalization

- NLP tasks perform 3 important text normalizations:
  1) segmenting / tokenizing words in running text
  2) normalizing word formats
  3) segmenting sentences in running text

2.5.1 Tokenization

- idea: define what we consider a unit of a text (i.e. "word") to consider
- also called **word segmentation**

Definitions

- **lemma**: two words have the same lemma if they have
  - same stem
  - same part of speech
  - same rough word sense
  - (e.g. "cats" and "cat")
- **wordform**: fully inflected surface form of a word
  - e.g. both "cats" and "cat" are different wordforms of the same lemma
- **type**: an element of the vocabulary V
- **token**: an instance of a type in running text
- **law of Church and Gale**

$$|V| > O\left(N^{\frac{1}{2}}\right) \tag{2.1}$$

  where
  - N: number of tokens

Common Problems

- one or two word problem: "San Francisco", "data base", "Hewlett-Packard"
- numbers
- segmentation problem / missing whitespace (mostly for languages like Inuit, etc.)
- multiple alphabets in same language (Japanese)
- different writing directions (Arabic)
  - text and numbers are written in different directions
- accents and umlauts
  - usually removed, but can be problematic

Maximum Matching Algorithm

- used e.g. for Chinese
- assumption: given a vocabulary of the language and a string to tokenize
- idea: always take longest word in the dictionary which matches the next sequence from the string

2.5.2 Normalization

- most common way: define **equivalence classes** of terms
  - can be done implicitly by removing characters like hyphens
  - BUT: only easy when removing characters (but not when adding chars)
- alternative: **asymmetric expansion**
  - examples
    - window → window, windows
    - windows → Windows, windows
    - Windows (no expansion)
  - idea: create expansion lists of different terms which can overlap **without being identical**
  - more powerful but less efficient than equivalence classes
- normalization and language detection interact (mit Hund vs. go to M.I.T)

Case Folding

- idea: reduce all letters to lower case
- only good in some applications:
  - IR
  - case is important for: sentiment analysis, machine translation and information extraction

2.5.3 Stemming and Lemmatization

- documents use different forms of words (e.g.: organize, organizing, organizes)
- overall goal of stemming and lemmatization: reduce inflectional form and sometimes derivationally related forms of a word to a common base form

Lemmatization

- process uses a vocabulary and morphological analysis of words
- aims at removing inflectional endings only in order to return the base form of a word (**lemma**)

Stemming

- crude **heuristic process** that **chops off end of words** (in the hope to be right most of the time)
- language dependent
- most common algorithm: **Porter's stemmer**
- in general: stemmer use language specific rules but require less knowledge than a lemmatizer
- stemming tends to be worse them lemmatization in languages with a lot of **morphology**

Porter's Stemmer

- empirically very effective
- consists of five phases of word reduction, applied sequentially
- in each phase: conventions to select rules

Morphology

- **morphemes**: small meaningful units that make up words
- two types:
  - **stems**: core meaning-bearing units (e.g. man, woman, health...)
  - **affixes**: bits and pieces that adhere to stems (e.g. -ly, -er, -ism, -ness, ...)
- morphological phenomena depend on language
- some languages require complex morpheme segmentation (e.g. Turkish)

2.6 Sentence Segmentation

- "!" and "?" are relatively clear symbols of ends of sentences
- "." is more problematic (e.g. abbreviations, numbers, etc.)

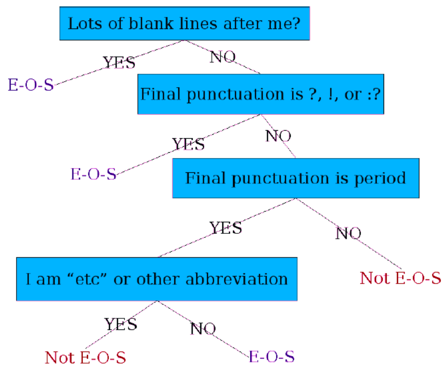
2.6.1 Idea

- idea: build a binary classifier, deciding if a given character marks the end of a sentence or not
- possible classifiers
    - hand-written rules
    - regular expressions
    - machine learned
        - e.g. decision trees

2.6.2 Decision Trees for Sentence Segmentation

Decision Tree

- each inner node represents a decision based on one attribute (i.e. feature)
- each leaf is a final decision (i.e. usually "yes" or "no")
- at query time: start at root and take decisions until a leaf node is reached ⇒ value is final decision



Building a Decision Tree

- can be constructed by experts (only for simple features and small trees)
- but are usually machine learned based on a training corpus
- choosing the features is the interesting part

Possible Features

- case of word before/after ".": upper, lower, cap and number
- numeric features:
    - length of word before "."
    - $\mathbb{P}$(word with "." occurs at end-of-sentence)
    - $\mathbb{P}$(word with "." occurs at beginning-of-sentence)

CHAPTER 3

(Probabilistic) Language Modeling

3.1 Probabilistic Language Models

- idea: model assigns a probability to a sentence (or sequence of words)

  $\Rightarrow$ allows to predict words based on preceding words

- applications:
  - machine translation
  - spell correction (which alternative has the higher probability?)
  - speech recognition
  - question and answering
  - etc.

3.1.1 Formal Definition

- language models compute either
  - probability of a sentence

$$\mathbb{P}(W) = \mathbb{P}(w_1 w_2 \dots w_n) \tag{3.1}$$

  - or probability of an upcoming word

$$\mathbb{P}(w_i | w_1, w_2 \dots w_{i-1}) \tag{3.2}$$

- note: a language model actually models grammar!

3.1.2 Computing Probabilities

- idea: compute $\mathbb{P}(w_1 w_2 \dots w_n)$ by applying the **chain rule of probability**

$$\begin{aligned}\mathbb{P}(w_1 w_2 \dots w_n) &= \mathbb{P}(w_1) \cdot \mathbb{P}(w_2|w_1) \cdot \mathbb{P}(w_3|w_1, w_2) \cdot \ldots \cdot \mathbb{P}(w_n|w_1, w_2, \dots, w_{n-1}) \\ &= \prod_{i=1}^{n} \mathbb{P}(w_i|w_1, \dots, w_{i-1})\end{aligned} \tag{3.3}$$

- valid approach in theory, BUT
- infeasible in practice due to huge amount of parameters
  - reliably estimating all probabilities would require a ridiculously huge amount of data $\Rightarrow$ not possible

$\Rightarrow$ **Makov assumption**

Markov Assumption

- idea: we can predict the probability of some future unit without looking too far into the past
- implementation:

$$\mathbb{P}(w_1 w_2 \dots w_n) \approx \prod_{i=1}^{n} \mathbb{P}(w_i|w_{i-\mathbf{k}}, \dots, w_{i-1}) \tag{3.4}$$

  where k can be seen as a parameter
- k determines how far we look into the past
- k is usually defined via **k-grams** (n-grams)

3.1.3 N-Grams Models

- unigram model - no look into the past

$$\mathbb{P}(w_1 w_2 \dots w_n) \approx \prod_{i=1}^{n} \mathbb{P}(w_i) \tag{3.5}$$

- bigram model - look one word into the past

$$\mathbb{P}(w_1 w_2 \dots w_n) \approx \prod_{i=1}^{n} \mathbb{P}(w_i|w_{i-1}) \tag{3.6}$$

- can be generalized to n-grams
- in general insufficient for language modeling
  - cannot capture **long-distance dependencies**
  - BUT often good enough

3.2 Estimating N-Gram Probabilities

- need for a training corpus
- mark start ($<s>$) and end ($</s>$) of sentence with special tokens
- for bigrams: take the count of a particular bigram, and divide this count by the sum of all the bigrams that share the same first word

$$\widehat{\mathbb{P}}(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i)}{\sum\limits_{w \in V} c(w_{i-1}, w)} = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})} \tag{3.7}$$

- for k-grams:

$$\widehat{\mathbb{P}}\left(w_i|w_{i-k+1}^{i-1}\right) = \frac{c\left(w_{i-k+1}^{i-1}, w_i\right)}{c\left(w_{i-k+1}^{i-1}\right)} \tag{3.8}$$

- divisor in the formulas above is for scaling to the unit interval

3.2.1 Maximum Likelihood Estimation

- estimates above are **relative frequencies**
- estimating probabilities based on relative frequencies is a **maximum likelihood estimation**
  - estimates are parameters of a model M
  - MLE maximizes the likelihood of the training set T given the model M (i.e. its parameters)
  - i.e. parameters are chosen such that the given data is the most likely one (compared to other data)

3.2.2 Practical Considerations

- perform multiplication of small probabilities in log-space
  - we can sum instead of multiply
  - is faster and avoids risks of underflow

3.3 Evaluation and Perplexity

- idea: evaluate if our language model prefers good to bad sentences
- two main methods:
  - extrinsic
  - intrinsic

3.3.1 Components

- **training dataset** (used to learn parameters of our model)
- **test dataset** (used to test the models performance)
  - needs to be untouched for training!
- **evaluation metric**

3.3.2 Extrinsic Evaluation

- perform a specific NLP task with each model you want to compare (e.g. spelling correction)
- measure accuracy of each model and rank them accordingly
- **problem**: very time-consuming

3.3.3 Intrinsic Evaluation (Perplexity)

- idea: play the Shannon game of predicting the next word / sentence
  - better model is the one which assigns the higher probability to the word that actually comes next
- formalization via **perplexity**
- **problem**: bad approximation (unless test data looks just like the training data)

Perplexity

- measure of how well a probability model predicts a sample
- **definition**: inverse probability of the test set (or a sentence of it) normalized by the number of words

$$PP(W) = \mathbb{P}(w_1 \dots w_n)^{-\frac{1}{n}} = \sqrt[n]{\frac{1}{\underbrace{\mathbb{P}(w_1 \dots w_n)}_{\text{approx: Markov}}}} \tag{3.9}$$

- minimizing perplexity is the same as maximizing probability
- i.e.: best model has minimal perplexity!
- perplexity of two models can only be compared if they use the same vocabulary
- perplexity improvement does not guarantee an extrinsic performance improvement
- perplexity can also be seen as a **weighted average branching factor**
  - model the prediction of a sentence as a tree
  - nodes represent decision points, edges are decisions which we take according to the probability assigned by the model

3.4 Model Visualization

Shannon Method

- assume we have a bigram model
- approach
  1) choose a random bigram $(< s >, w)$ according to its probability (where $w_0 \in V$ is any word)
  2) choose a random bigram (w, x) according to its probability $(x \in V)$
  3) continue until we choose $< /s >$
  4) concatenate bigrams in order together (removing duplicate words due to bigram overlap)

I want to eat Chinese food
$< s >$**I**
I want
want to
to eat
eat Chinese
Chinese food
food$< /s >$

3.5 Generalization and Zeros

3.5.1 Dangers of Overfitting

- n-gram models only word well for word prediction of the test corpus looks like the training corpus (i.e. we overfit very easily)
- one option to train a more robust model (i.e. generalizes better): **smoothing** out 0s
  - problem: we might not have seen bigrams from the test set in the training set

  ⇒ 0 probabilities

  - when trying to compute the probability of a sentence containing an unseen bigram, the product becomes 0

  ⇒ perplexity cannot be computed (division by 0)

3.5.2 Smoothing: Laplace / Add-One

- idea: add one to all counts
- for bigrams:
$$\widehat{\mathbb{P}}(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + |V|} \tag{3.10}$$
- **problems**:
  - variances of counts is worse than the unsmoothed counts
  - much worse at predicting the actual probability than other methods for smoothing
- not used for n-grams anymore, BUT okay for other applications
  - text classification (predicting the right class, not the exact probability is important!)
- alternative view on concept: we **discount** non-zero values

3.5.3 Smoothing: Absolute Discounting

- idea: substract probability mass from bigrams with a certain count to save probability mass for the zeros
- amount to substract: can be learned by looking at the difference in counts between a training and held-out set
  - BUT: very time consuming
  - easier: just substract some d (e.g. $d = 0.75$)
- **approach**: create standard model on training set and new model for held-out set based on original one, where
$$\mathbb{P}_{\text{AbsDisc}}(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i) - d}{c(w_{i-1})} + \lambda(w_{i-1}) \cdot \mathbb{P}(w_i) \tag{3.11}$$
where
  - $\lambda(w_{i-1})$: interpolation weight
- problem: $\mathbb{P}(w_i)$ is not a good estimate for lower-order unigram distributions ⇒ Kneser-Ney smoothing

3.5.4 Smoothing: Kneser-Ney

- better unigram approximation: $\mathbb{P}_{\text{continuation}}(w)$ - how likely is w to appear as a novel continuation
  - estimation: for each word, count the number of bigram types it completes
$$\mathbb{P}_{\text{continuation}}(w) \propto |\{w_{i-1}|c(w_{i-1}, w) > 0\}| \tag{3.12}$$
  - normalize by total number of bigram types (to obtain a probability)
$$\mathbb{P}_{\text{continuation}}(w) \propto \frac{|\{w_{i-1}|c(w_{i-1}, w) > 0\}|}{|\{w_{j-1}|c(w_{j-1}, w_j) > 0\}|} \tag{3.13}$$

⇒ frequent words occurring in only **one or few contexts**: **low continuation probability**

- **final** Kneser-Ney for bi-grams

$$\mathbb{P}_{\text{KN}}(w_i|w_{i-1}) = \frac{\max\left(c(w_{i-1}, w_i) - d, 0\right)}{c(w_{i-1})} + \lambda(w_{i-1}) \cdot \mathbb{P}_{\text{continuation}}(w_i) \tag{3.14}$$

where λ is a normalizing constant (i.e. the probability mass we have discounted)

$$\lambda(w_{i-1}) = \frac{d}{c(w_{i-1})} \cdot |\{w|c(w_{i-1}, w) > 0\}| \tag{3.15}$$

3.5.5 Backoff and Interpolation

- idea: if we have 0-counts (i.e. few knowledge about certain n-grams), we can **condition on less context** (in the hope to have more knowledge about the smaller context)
- two strategies:
  - **backoff**: use n-gram if we have enough evidence, otherwise $n-1$-gram, etc..
  - **interpolation**: mix different n-gram models (with different n)
- also a form of smoothing

3.5.6 Linear Interpolation

Simple Interpolation

- tri-gram example:

$$\widehat{\mathbb{P}}_\lambda(w_i|w_{i-2}, w_{i-1}) = \lambda_1 \cdot \widehat{\mathbb{P}}(w_i|w_{i-2}, w_{i-1}) + \lambda_2 \cdot \widehat{\mathbb{P}}(w_i|w_{i-1}) + \lambda_3 \cdot \widehat{\mathbb{P}}(w_i) \tag{3.16}$$

where $\sum_i \lambda_i = 1$

Lambdas Conditional on Context Interpolation

- idea: make weights (λ dependent on context of the original tri-gram
- weights $\lambda_k(w_{i-2}, w_{i-1})$ is essentially now a function defining the weight for a given context (words w_{i-2}, w_{i-1})
- tri-gram example:

$$\begin{aligned} \widehat{\mathbb{P}}_\lambda(w_i|w_{i-2}, w_{i-1}) = {} & \lambda_1(w_{i-2}, w_{i-1}) \cdot \widehat{\mathbb{P}}(w_i|w_{i-2}, w_{i-1}) \\ & + \lambda_2(w_{i-2}, w_{i-1}) \cdot \widehat{\mathbb{P}}(w_i|w_{i-1}) \\ & + \lambda_3(w_{i-2}, w_{i-1}) \cdot \widehat{\mathbb{P}}(w_i) \end{aligned} \tag{3.17}$$

where $\sum_k \lambda_k = 1$

- advantage: since we have one parameter per bi-gram now, we can increase weights for bi-grams with more evidence

- both versions:
  - separate from actual training corpus used for learning the probability estimates
  - choose λ_k s.t. the likelihood of the held-out corpus is maximized

Learning λ Parameters

- assume we have training data, held-out data (i.e. smaller training set, but separate from it) and test data
- process
  1) estimate probabilities for n-grams based on training data
  2) learn parameters λ_k which maximize the likelihood on the held-out data (with the fixed estimates from above):

$$\arg\max_{\lambda} \log\left(\widehat{\mathbb{P}}_{\lambda}\left(w_1, \ldots, w_n\right)\right) \tag{3.18}$$

     where $w_1, \ldots, w_n$ is the held-out data

3.5.7 Dealing with Unknown Words

- two kinds of tasks
  - **closed vocabulary task**: vocabulary V is fixed from the beginning
  - **open vocabulary task**: vocabulary is not fixed
- unknown words are called **out of vocabulary words** (OOV words)
- can be handled by creating an **unknown word token** *<UNK>*
- estimating *<UNK>* probabilities:
  1) create a fixed lexicon L of size V (by deleting all words under a certain threshold wrt frequency)
  2) in normalization phase: change any word not in V to *<UNK>*
  3) estimate probabilities as usual
- during model usage: treat unknown words as *<UNK>*

3.5.8 Huge Web-Scale n-Gram Corpora

- for huge web-scale n-Gram corpora, we have to be very efficient
- use pruning
  - only store n-grams with a frequency over a threshold
  - entropy (i.e. relevance) based pruning
- consider efficiency
  - efficient datastructures

- bloom filters: approximate language models
- store words as indexes

Smoothing for Web-Scale n-Grams

- **stupid backoff**
- no discouting - just use relative frequencies
- S instead of $\mathbb{P}$ as symbol for k-gram-probability

$$S(w_i|w_{i-k+1}^{i-1}) = \begin{cases} \frac{c(w_{i-k+1}^{i})}{c(w_{i-k+1}^{i-1})} & \text{if } c(w_{i-k+1}^{i}) > 0 \\ 0.4 \cdot S(w_i|w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases} \tag{3.19}$$

  where

$$S(w_i) = \frac{c(w_i)}{T} \tag{3.20}$$

  where T: number of tokens in corpus
- **note**: NOT a true probability distribution anymore

3.6 Advanced Language Models

- **discriminative models**
  - choose n-gram weights to improve a task not to fit the training set
- parsing-based models
- caching models
  - recently used words are more likely to appear

CHAPTER 4

Spell Checking

4.1 Error Classification

two types or errors

- **non-word errors**
  - words which do not exist in reference language
- **(real-)word errors**
  - words which exist in reference language but are wrong in the context (flew *form* Heathrow)
  - two subtypes:
    - typos
    - cognitive errors (piece vs peace)

4.1.1 Associated Problems

1) non-word error detection
2) isolated-word error correction
   - mapping a non-word error to the correct word
3) context-dependent error detection and correction
   - using context to detect and correct non-and real-word errors

Non-Word Error Detection

- any word not in a **dictionary** is considered an error
- size of dictionary
  - original approach: keep it small (in order to detect common spelling errors involving uncommon words)
  - BUT: large dictionaries are more helpful
    - precision is increase as words detected as errors are more likely true errors
    - recall is decreased as we might miss some errors

Non-Word Error Correction

- given a word, generate set of **candidates** (correct words similar to error)
  - similar wrt. e.g. edit distance, soundex, etc.
- rank according to some measure (e.g. edit distance) among candidates

Real-Word Error Correction

- problem: we do not know if a word is wrong or not
- solution:
  1) for each word w in the sentence, generate a candidate set as above (including w)
  2) let a method (e.g. classifier) choose the best candidate separately or looking at all combinations of words

4.2 Distance Measures

4.2.1 Edit Distance

- distance between string s_1 and s_2 is the minimum number of basic operations required to transform s_1 into s_2
- example implementations:
  - **Levenshtein** basic operations: insert, delete and replace
  - **Damerau-Levenshtein** basic operations: additionally transposition

```
LevenshteinDistance(s1, s2)
for i ← 0 to |s1|
do m[i, 0] = i
for j ← 0 to |s2|
do m[0, j] = j
for i ← 1 to |s1|
do for j ← 1 to |s2|
    do if s1[i] = s2[j]
          then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
          else  m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
return m[|s1|, |s2|]
```

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

- [**ToDo**: *add annotations in algo (which operation is which equation part*]
- how to read matrix (to get operations):
  1) start at bottom right
  2) at each step:
     a) look which sub cell is minimum (at tie: select any)
     b) note down the action corresponding to it
     c) go tht ecell belonging to the winning sub-cell
  3) reverse order of actions

  - note: for "replace or copy" you have to check if the cost increased
  - left side of matrix: input, right side of matrix: output

cost of getting here from my upper left neighbor (copy or replace)	cost of getting here from my upper neighbor (delete)
cost of getting here from my left neighbor (insert)	the minimum of the three possible “movements”; the cheapest way of getting here

		s	n	o	w
	0	1 1	2 2	3 3	4 4
o	1 1	1 2 2 1	2 3 2 2	2 4 3 2	4 5 3 3
s	2 2	1 2 3 1	2 3 2 2	3 3 3 3	3 4 4 3
l	3 3	3 2 4 2	2 3 3 2	3 4 3 3	4 4 4 4
o	4 4	4 3 5 3	3 3 4 3	2 4 4 2	4 5 3 3

cost	operation	input	output
1	delete	o	*
0	(copy)	s	s
1	replace	l	n
0	(copy)	o	o
1	insert	*	w

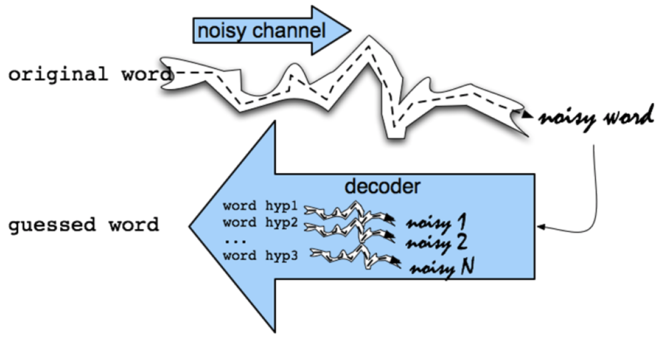
- runtime: $O(|s_1| \cdot |s_2|)$
- **dynamic programming approach**
  - optimal substructures: optimal solution contais subsolutions
  - subsolutions overlap: are computed over and over again when using a brute-force algorithm
  - **subproblem for edit distance**: edit distance of two prefixes
  - **overlap for edit distance**: distances of prefixes are required 3 times
- variant: **weighted edit distance**
  - weight of an operations depends on characters involved
  - used for keyboard errors ("m" more likely to be mistyped as "n" than as "q")

4.2.2 Soundex (Distance)

- represent words in a 4-char reduced form (signature) s.t. **homophones** (word which sound similar) are encoded in a similar fashion
- compute distance wrt. signatures instead of strings (e.g. using edit distance)
- several variants of original soundex for different languages

4.3 Noisy Channel Model

- idea: original words go through a **noisy channel**, become a **noisy word** from which the real word must be guessed by **decoder**



4.3.1 Concept

- given an observation x of a misspelled word
- approximate correct word by

$$\widehat{w} = \arg\max_{v \in V} \mathbb{P}(v|x) \tag{4.1}$$

- problem: hard to estimate probabilities, as x might have never been seen before!
- solution: **Bayes rule**!

$$\widehat{w} = \arg\max_{v \in V} \mathbb{P}(v|x) = \arg\max_{v \in V} \frac{\mathbb{P}(x|v) \cdot \mathbb{P}(v)}{\mathbb{P}(x)} = \arg\max_{v \in V} \mathbb{P}(x|v) \cdot \mathbb{P}(v) \tag{4.2}$$

  - last step is possible, as $\mathbb{P}(v)$ is **constant** for all $v \in V$

4.3.2 Complete Approach

1) given an incorrect word x, generate a set of candidates C (as above)
2) return $v \in C$ for which holds

$$\arg\max_{v \in C} \mathbb{P}(x|v) \cdot \mathbb{P}(v) \tag{4.3}$$

Remarks

- using C instead of V is empirically good idea
- 80% of errors are within edit distance 1 and almost all in distance 2
- (if we allow insertion of **space** and **hyphen** as well

⇒ processing the whole vocabulary would be a waste!

4.3.3 Probability Estimation

Prior Probability Estimation

- estimating $\mathbb{P}(v)$
- count frequency in large corpus (10^7+ words) (maybe including +1 smoothing)

Likelihood / Channel Probability Estimation

- $\mathbb{P}(x|v)$
- in general: unsolved problem (depends very much writer and many other factors)
- approximate solution: estimate $\mathbb{P}(x|v)$ by the **probability of edit**

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

where

- x_i is the ith character of the misspelled word x
- w_i is the ith character of a correct word w

$$\begin{aligned} &\text{del}[x,y]\text{: count}(\mathbf{xy} \text{ typed as } \mathbf{x}) \\ &\text{ins}[x,y]\text{: count}(\mathbf{x} \text{ typed as } \mathbf{xy}) \\ &\text{sub}[x,y]\text{: count}(\mathbf{x} \text{ typed as } \mathbf{y}) \\ &\text{trans}[x,y]\text{: count}(\mathbf{xy} \text{ typed as } \mathbf{yx}) \end{aligned}$$

- assumption: each misspelling of a word can be reached via one of the 4 confusion matrices
  - plausible as most typos involve one 1 or 2 characters
- counts can be obtained from confusion matrices (one for each edit operation)
- confusion matrices computed based on some large dataset of errors and corrections

4.3.4 Improved Channel Model

- channel model can be used to create a ranking on the candidates
- take top-k elements and re-rank using using a given language model

⇒ this way we can use surrounding words for more context

4.3.5 Implicit Assumptions

- implicit assumption: **independence between words in sentence**
  - of course not true

⇒ add weighting scheme to prior and likelihood multiplication

$$\arg\max_{v \in C} \mathbb{P}(x|v) \cdot \mathbb{P}(v)^{\lambda} \tag{4.4}$$

where λ is a weight learned on a development test set

4.4 Evaluation

- measures
  - precision/recall/f-measure of detection
  - suggestion quality (correct word is first suggestion? correct word is in top-n suggestions?)

4.5 Real-Word Spelling Correction

- 25-40 % of spelling errors are real word errors

4.5.1 Basic Approach

1) generate candidate set
2) choose best candidate
   - via noisy channel model OR
   - task-specific classifier

4.5.2 Noisy Channel Model Approach

1) given a sentence $w_1, w_2, \ldots, w_n$
2) generate a set of candidates for each word w_i
3) choose sequence W maximizing $\mathbb{P}(W)$
   - note: we choose the sequence from the set of all possible sequences based on the candidate sets

Problem

- huge number of alternative sequences
  - assume k alternatives per candidate set and n words in the sentence

  ⇒ k^n sequences!

Simplification: One Error per Sentence

- instead of choosing each word out of the associated candidates, we generate sequences where only ONE word differs from the original one

⇒ $n\dot{k}$ sequences

Channel Model Adaptation

- for real-word errors, channel model has to additionally consider the **probability of no error** $\mathbb{P}(w|w)$
- $\mathbb{P}(w|w)$ can be estimated by counting the error frequency in a corpus (including smoothing!)

4.6 Further Considerations

4.6.1 Showing Errors to the User

- the more confident we are about an error, the more drastic we can be about the correction
- e.g.: very confident $\Rightarrow$ autocorrect
- e.g.: unconfident $\Rightarrow$ flag as potential error

4.6.2 Classifier Based Methods

- instead of just channel model and language model
- use many features in a classifier
- build classifier deciding between common misspellings: e.g. (whether vs. weather)

CHAPTER 5

Deduplication

5.1 Introduction

- idea: given a large number of documents, find *near* duplicate pairs

5.1.1 Formal Definition

- given
  - set of documents D
  - similarity function $\sigma : D \times D \to \mathbb{R}$
  - similarity threshold $\phi \in \mathbb{R}$
- goal: find

$$M = \{(d, d') \in D^2 | \sigma(d, d') \geq \phi\} \tag{5.1}$$

5.1.2 Applications

- crawling (do not crawl duplicates)
- (approximate) mirror pages (do not show both in search results)
- similar news articles
- fact checking (near duplicates contain similar facts)

5.1.3 Issues for Recognizing Duplicates

- many small pieces of one document can appear out of order in another
  - e.g. due to plagiarism

  $\Rightarrow$ need for **robust similarity function** σ
- too man documents
  - naive approach: compare all pairs of documents $\Rightarrow O(|D|^2)$

  - $O(|D|^2)$ is too long in practice!
- documents are so large or so many that they cannot fit into main memory

  $\Rightarrow$ need for **compressed representation**

5.1.4 Similarity Functions

- Damerau-Levenshtein distance (negative)
- Soundex
- **Jaccard similarity**

$$Jaccard(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} \in [0,1] \tag{5.2}$$

  where X, Y are sets

5.2 Deduplication with Jaccard

5.2.1 Approach Overview

1) **Shingling**: convert documents to sets (of k-shingles)
   - sets of strings of length k to describe documents
2) **MinHashing**: convert large sets to short signatures, while preserving similarity
   - integer vectors to represent sets
   - vectors encapsulate similarity
3) **Locality-Sensitive Hashing (LSH)**: instead of comparing all pairs of documents, find possibly similar documents by hashing documents based on their signatures
   - only compare actual similarity on candidate pairs found by LSH

5.2.2 Shingling

- idea: convert documents to sets
- simplest approach: represent document by set of words it contains
  - problem: we do not account in any way for the ordering of words
  - solution: **shingles** $\Rightarrow$ represent a document by the set (or multiset) of shingles it contains
- **k-shingle** (or k-gram): sequence of k tokens
  - token: can be a character, a word or something else
  - we assume: token = character
- **working assumption**: documents which have many shingles in common have similar text (even if text appears in different order)
- **important**: k must be large enough!

- if k is too small, we have a huge number of candidate pairs (as the e.g. same 2-shingles appear in almost any document)
- rule of thumb
  - short documents: $k \approx 5$
  - long documents: $k \approx 10$

Efficient Representation

- explicit set representation is not well suited for rest of deduplication process
- solution: represent a set as **binary vector** $d \in \{0,1\}^{|\mathcal{S}|}$ in the space of k-shingles $\mathcal{S}$ where

$$d_i = \begin{cases} 1 & \text{if shingle } i \text{ is present in set} \\ 0 & \text{else} \end{cases} \tag{5.3}$$

⇒ vectors are very sparse

- important operations
  - set intersection: bitiwse AND (of two vectors)
  - set union: bitwise OR (of two vectors)
  - e.g. $d = 10111$, $d' = 10011 \Rightarrow d \cap d' = 10011$ and $d \cup c' = 10111$

5.2.3 MinHashing

- goal: convert large sets to short signatures (e.g. hashes) while preserving similarity
- **goal**: find hash function h such that
  1) if $\sigma(d, d')$ is high $\Rightarrow h(d) = h(d')$ with high probability
  2) if $\sigma(d, d')$ is low $\Rightarrow h(d) \neq h(d')$ with high probability

⇒ compare similarity of signatures instead of original document vectors!

From Sets to Boolean Matrices

- we can place the vector representation of all documents $d \in D$ in a matrix
  - each document vector is a **column**

  ⇒ each row represents a shingle

⇒ matrix is very sparse

- 4 documents (columns)
- 7 singles (rows)

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

- **column similarity** (between two columns): Jaccard similarity of the corresponding sets

Signature / Hash Function Computation

- idea: compute signature of column by looking at several row permutations of boolean matrix
- let π be a row permutation
- let $h_\pi(d)$ = index of first row in which column of d has a 1 (under row permutation π)

$$h_\pi(d) = \min\{i | d_{\pi(i)} = 1\} \tag{5.4}$$

- **signature**:
  - use several **independent** hash functions (i.e. permutations) and aggregate their values as a vector (for a column)

$\Rightarrow$ dimension of signature vector = number of hash functions

- **signature matrix** M:
  - M is a $K \times |D|$ matrix, where K is the number of hash functions
  - M each column in M represents the MinHash signature of the corresponding document
- example: 4 docs, 2 permutations, i.e. hash functions $\Rightarrow$ dimension of signatures: 2

$$D = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \qquad \Pi = \begin{bmatrix} 2 & 4 \\ 3 & 2 \\ 7 & 1 \\ 6 & 3 \\ 5 & 6 \\ 1 & 7 \\ 4 & 5 \end{bmatrix} \qquad sig(D) = \begin{bmatrix} 1 & 1 & 3 & 1 \\ 2 & 2 & 3 & 1 \end{bmatrix}$$

- in practice: the more hash functions, the better

MinHash Property

- assume we pick the document vector representations at random
- for any permutation π it holds that

$$\mathbb{P}\left(h_\pi(d) = h_\pi(d')\right) = \sigma(d, d') \tag{5.5}$$

- proof

- let $y \in X$ be a shingle where X is the set of shingles (we assume we represent shingles by a number)
- it holds that
$$\mathbb{P}(y = \min(\pi(X))) = \frac{1}{X} \tag{5.6}$$
- let y be s.t. $\pi(y) = \min(\pi(d \cup d'))$
- then
  1) $\pi(y) = \min(\pi(d))$ OR
  2) $\pi(y) = \min(\pi(d'))$
- probability that both events are true, i.e. $\mathbb{P}(\min(\pi(d)) = \pi(y) = \min(\pi(d')))$, is equivalent to
$$\mathbb{P}(y \in d \cap d') = \frac{|d \cap d'|}{|d \cup d'|} \tag{5.7}$$
- hence
$$\begin{aligned} \mathbb{P}(h_\pi(d)) = h_\pi(d')) &= \mathbb{P}(\min(\pi(d)) = \min(\pi(d'))) \\ &= \mathbb{P}(y \in d \cap d') \\ &= \frac{|d \cap d'|}{|d \cup d'|} = \sigma(d, d') \end{aligned} \tag{5.8}$$

Similarity of Signatures

- **similarity of signatures**: fraction of the hash functions for which the signatures agree
- expected similarity of signatures = similarity of vector representations of documents

Practical Considerations

- the more hash functions ⇒ the better
- signature of document is usually a lot smaller than the original vector

5.2.4 Locality Sensitive Hashing

Idea

- we want to avoid computing the similarity between all pairs of documents

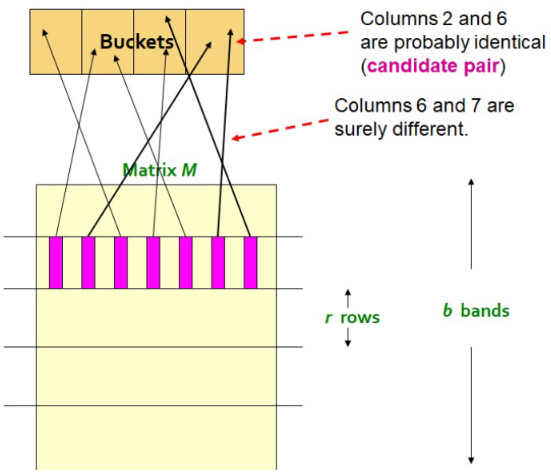
⇒ we only compare the actual similarity of **candidate** document-pairs

⇒ find a function $f : D \times D \to \{\mathit{true}, \mathit{false}\}$ telling whether two given documents resemble a candidate pair

- for MinHash matrices:
  1) hash columns of signature matrix to many buckets
  2) pairs of documents in the same bucket are candidate pairs

Approach

1) divide matrix M into b bands, each of r rows
   - hence: $K = b \cdot r$ (K: signature size)
2) for each band: hash its part of each column (i.e. a partial document) to a hash table with κ buckets
   - κ should be as large as possible to avoid accidental collisions
3) candidate document pairs : those which hash to the same bucket for ≥ 1 band(s)



- Suppose $N = 10^5$, ergo M has 10^5 columns
- Signature size $K = 100$
- Choose $b = 20$ and $r = ?$ 5
- Goal: Find pairs of documents with $\sigma \geq 0.8$

If $\sigma(d_1, d_2) = 0.8$

- Probability that they share a common signature $= (0.8)^5 \approx 0.326$
- Probability that they are not similar in any of the bands $= ?$ $(1 - 0.326)^{20} \approx 0.00035$
- What is our expected recall? 99.965%

$\Rightarrow$ We have false negatives

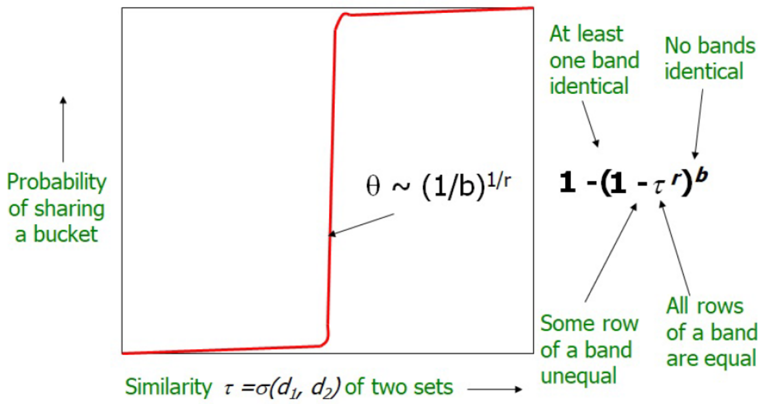
Parameter Tuning Considerations

- parameters b are r need to be tuned s.t.
  1) most of the similar document pairs are found
  2) and only few non-similar document pairs are found
- note: the fewer bands, the fewer false positives, but also more false negatives
- parameter K (signature length) can also be tuned

Approach Analysis

- let $\sigma(d, d') = \tau$
- for any band (r rows):
  - prob. that all rows in band are equal: τ^r
  - prob. that any row in the band is not equal to the others: $1 - \tau^r$
  - prob. that no band is identical: $(1 - \tau^r)^b$
  - prob. that at least one band is identical: $(1 - (1 - \tau^r)^b)$
- function $\tau \to (1 - (1 - \tau^r)^b)$ is sigmoid
  - function is (roughly) 1/2 for $\tau = \left(\frac{1}{b}\right)^{1/r}$ (for large b and r) $\Rightarrow$ optimal value without further application knowledge

  $\Rightarrow$ in general: try to set b and r s.t. $\theta = \left(\frac{1}{b}\right)^{1/r}$
  - for specific applications it might make more sense to shift the value into one of the two directions (e.g. lower threshold to avoid false positives)



Remarks

- LSH has the risk of BOTH false negatives and false positives!

CHAPTER 6

Text Classification

6.1 Applications

- spam detection
- authorship identification
- language identification
- sentiment analysis

6.2 Classification Methods

6.2.1 Rule Based Classification

- complex rules determine the class of a given document
- very high accuracy if rules are refined carefully over time by an expert
- building and maintaining rather complicated and expensive
- (e.g. used by Google Alerts)

6.2.2 Statistical/Probabilistic Classification

- see classification as a machine learning problem
- use supervised learning to learn a classifier
- no free lunch: requires hand-classified training data
- BUT: this manual classification can be done by none-experts
- e.g.: Naive Bayes

6.3 Formal Definition

- given:
  - **document space** $\mathcal{X}$: vector space which is used to represent documents
  - fixed **set of classes** $\mathcal{C} = \{c_1, \ldots, c_j\}$
  - a **training dataset** $\mathcal{D} \subseteq \mathcal{X} \times \mathcal{C}$ consisting of entries (d, c), representing a document labeled with its class
    - data should be identically independently distributed (i.i.d assumption)
- idea:
  - use learning algorithm to learn a model $h : \mathcal{X} \to \mathcal{C}$ predicting the correct class for given document (representation)

6.4 k-Nearest Neighbor Classifier

6.4.1 Idea

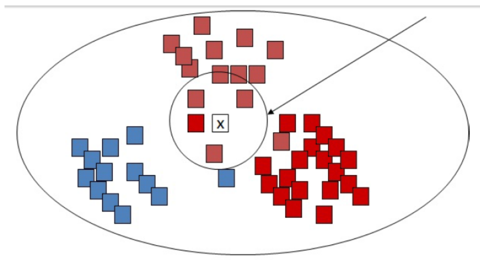
- given a document vector d, find the k nearest neighbors $\mathcal{K}(d) \subseteq \mathcal{D}$ wrt. some distance (in the training set)
- classify d as class c which is the most common class among those k neighbors
  - we can also use a probability interpretation according to number of times a class is found in the neighbor set
  - i.e. $\mathbb{P}(c|d) = \dfrac{|\mathcal{K}_c|}{k}$ ($\mathcal{K}_c$: number of neighbors with class c in $\mathcal{K}(d)$

6.4.2 Learning k and Importance of k

- k acts as a **smoothing factor**
  - with small k: high chance to overfit due to small sample size (high chance for outliers to influence result)
  - larger k: lower change of overfitting due to larger sample size
- optimize k via **cross-validation**
  - idea: split data into training, validation and test set
  - choose k which maximizes target measure on validation set
  - simplest form **leave-1-out cross-validation**
    - train on all but one data points and validate on the remaining one
    - repeat for all possible combinations of training and validation dataset and aggregate results (e.g. sum)
    - choose k with best performance on aggregated results

6.4.3 Probabilistic Interpretation

- basic idea: given a document d create a hyperball of volume V centered around d covering k points



- hyperball contains k_j points from class c_j
- N_j is the total number of points from class c_j in the whole training data
- $N = \sum_{j=1}^{|\mathcal{C}|} N_j$ is the total number of points in the training data
- **unconditional density around** x

$$\mathbb{P}(x) = \frac{k}{N \cdot V} \tag{6.1}$$

- **conditional density around** x

$$\mathbb{P}(x|c_j) = \frac{k_j}{N_j \cdot V} \tag{6.2}$$

- with **Bayes rule**

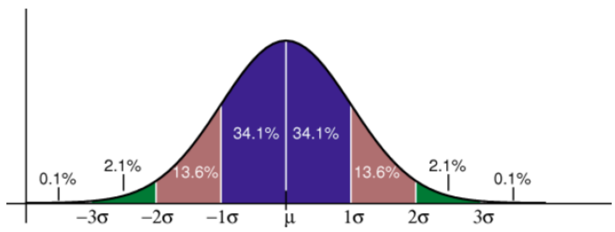
$$\mathbb{P}(c_j|x) = \frac{\mathbb{P}(x|c_j) \cdot \mathbb{P}(c_j)}{\mathbb{P}(x)} = \frac{k_j}{k} \tag{6.3}$$

as $\mathbb{P}(c_j) = \frac{N_j}{N}$

6.4.4 Weighted (Probabilistic) kNN

- idea: **weight the contribution** of each neighbor **based on distance** to the point of interest
- (Gaussian) weight function motivated by Gaussian distribution

$$w(\mathbf{x}, \mathbf{x}_i) = exp(-\lambda ||x - x_i||_2^2) \text{ with } \lambda > 0$$



- prediction: return class according to

$$\arg\max_{c\in\mathcal{C}} \mathbb{P}(c|x) = \frac{\sum\limits_{(x',c')\in\mathcal{K}(x)} w(x,x')\cdot [\![c'=c]\!]}{\sum\limits_{(x',c')\in\mathcal{K}(x)} w(x,x')} \tag{6.4}$$

Learning λ

- learn using cross-validation (e.g. leave-1-out cross-validation)

⇒ look for λ maximizing the **total prediction accuracy** (over all validation folds)

⇒ look for λ maximizing the (log)likelihood of the given data (i.e. maximize the probability that the left-out point in each fold gets assigned the correct class)

$$\begin{aligned}
\lambda^* &= \arg\max_{\lambda>0} \mathcal{L} \\
&= \arg\max_{\lambda>0} \sum_{j=1}^{|\mathcal{D}|} \log\left(\mathbb{P}(c_j|x_j,\mathcal{D}_j)\right) \\
&= \arg\max_{\lambda>0} \sum_{j=1}^{|\mathcal{D}|} \left(\log\left(\sum_{(x_i,c_i)\in\mathcal{K}(x_j)}^{x_i\neq x_j} w(x_j,x_i)\cdot [\![c_i=c_j]\!]\right) - \log\left(\sum_{(x_i,c_i)\in\mathcal{K}(x_j)}^{x_i\neq x_j} w(x_j,x_i)\right)\right) \\
&= \arg\max_{\lambda>0} \sum_{j=1}^{|\mathcal{D}|} \left(\log\left(\sum_{(x_i,c_i)\in\mathcal{K}(x_j)}^{x_i\neq x_j} \exp(-\lambda\cdot \parallel x_j - x_i \parallel_2^2)\cdot [\![c_i=c_j]\!]\right)\right) \\
&- \sum_{j=1}^{|\mathcal{D}|} \left(\log\left(\sum_{(x_i,c_i)\in\mathcal{K}(x_j)}^{x_i\neq x_j} \exp(-\lambda\cdot \parallel x_j - x_i \parallel_2^2)\right)\right)
\end{aligned} \tag{6.5}$$

where $\mathcal{D}_j$: dataset $\mathcal{D}$ except example (x_j, c_j)

⇒ $\mathcal{L}$ is difference of two **convex** functions

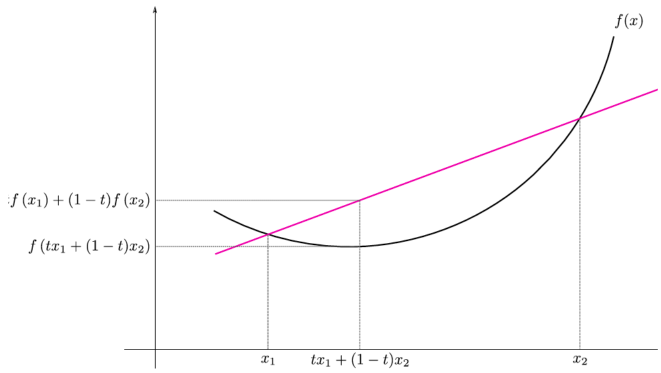
⇒ gradient descent/ascent works well for finding the minimum/maximum

Convex Function

- $f: X \to \mathbb{R}$ is **convex** if $\forall x_1, x_2 \forall t \in [0,1]$ it holds that

$$f(t\cdot x_1 + (1-t)\cdot x_2) \leq t\cdot f(x_1) + (1-t)\cdot f(x_2) \tag{6.6}$$

- roughly speaking: if we take any two points of the function and draw a line between them, the complete line is **above or on** the function



6.4.5 Advantages and Disadvantages

- advantages
  1) easy to implement (use efficient datastructure for searching nearest neighbors, e.g. R-tree)
  2) training is fast (mainly setting up the datastructure)
  3) learns complex target functions
- disadvantages
  - slow at query time (need find nearest neighbor)
  - easily fooled by irrelevant features ($\leq$ 20 features recommended)
  - usually large amounts of training data needed

6.5 The Naive Bayes Classifier

Idea

- represent documents as set of words

$\Rightarrow$ we use a **bag-of-words** model as we ignore the positions of the words

6.5.1 Concept

- compute probability of document d being in class c as

$$\mathbb{P}(c|d) \propto \mathbb{P}(c) \cdot \prod_{1\leq k\leq n_d} \mathbb{P}(d|c) \tag{6.7}$$

  where

  - n_d: number of tokens in the document
  - $\mathbb{P}(t_k|c)$: conditional probability of term t_k occurring in a document of class c
    - measure of how much **evidence** t_k contributes that c is the correct class
    - if evidence is identical for all terms, $P(c)$ becomes the only important variable

- $\mathbb{P}(c)$: prior probability of class c

- return best class according to **maximum a posteriori (MAP)** principle:

$$\arg\max_{c\in\mathcal{C}} \widehat{\mathbb{P}}(c|d) = \arg\max_{c\in\mathcal{C}} \widehat{\mathbb{P}}(c) \cdot \prod_{1\leq k\leq n_d} \widehat{\mathbb{P}}(t_k|c) \tag{6.8}$$

where all quantities with a hat are **estimates**

6.5.2 Practical Considerations

- computing the product can be arithmetically problematic (underflow when multiplying small probabilities)

- since log is a monotonic function, we can also take the log of the product without changing the ranking

⇒ in practice we compute

$$\arg\max_{c\in\mathcal{C}} \log\left(\widehat{\mathbb{P}}(c) \cdot \prod_{1\leq k\leq n_d} \widehat{\mathbb{P}}(t_k|c)\right) = \arg\max_{c\in\mathcal{C}} \log\left(\widehat{\mathbb{P}}(c)\right) + \sum_{1\leq k\leq n_d} \log\left(\widehat{\mathbb{P}}(t_k|c)\right) \tag{6.9}$$

- interpretation
  - each $\log\left(\widehat{\mathbb{P}}(c)\right)$ is a weight indicating the relative frequency of class c
  - complete term is measure of how much evidence there is for the document being in the class

6.5.3 Parameter Estimation

- prior:

$$\widehat{\mathbb{P}}(c) = \frac{N_c}{N} \tag{6.10}$$

  where
  - N_c: number of docs in class c
  - N: total number of docs

- conditional probabilities:

$$\widehat{\mathbb{P}}(t|c) = \frac{T_{c,t}}{\sum_{t'\in V} T_{c,t'}} \tag{6.11}$$

  where
  - $T_{c,t}$: number of tokens of t in training docs of class c (counting multiple occurrences!)

- we use a **Naive Bayes assumption** here: we assume $\widehat{\mathbb{P}}(t|c)$ is independent of the terms position in the document!

Problems with 0s

- when one term of the product / log sum is 0, the whole probability gets 0 / is undefined!
  - happens easily if a vocabulary term t does not occur in any document of class c

  $\Rightarrow$ we would never assign a document containing term t to c
- solution: **smoothing**
  - simplest form: **add-1-smoothing**

$$\widehat{\mathbb{P}}(t|c) = \frac{T_{c,t}+\mathbf{1}}{\sum\limits_{t' \in V}(T_{c,t'}+\mathbf{1})} = \frac{T_{c,t}+\mathbf{1}}{\left(\sum\limits_{t' \in V} T_{c,t'}\right) + |V|} \tag{6.12}$$

6.5.4 Time Complexity

mode	time complexity
training	$\Theta(\lvert\mathbb{D}\rvert L_{\mathrm{ave}} + \lvert\mathbb{C}\rvert\lvert V\rvert)$
testing	$\Theta(L_{\mathrm{a}} + \lvert\mathbb{C}\rvert M_{\mathrm{a}}) = \Theta(\lvert\mathbb{C}\rvert M_{\mathrm{a}})$

- L_{ave}: average length of a training doc, L_{a}: length of the test doc, M_{a}: number of distinct terms in the test doc, $\mathbb{D}$: training set, V: vocabulary, $\mathbb{C}$: set of classes
- $\Theta(|\mathbb{D}|L_{\mathrm{ave}})$ is the time it takes to compute all counts.
- $\Theta(|\mathbb{C}||V|)$ is the time it takes to compute the parameters from the counts.
- Generally: $|\mathbb{C}||V| < |\mathbb{D}|L_{\mathrm{ave}}$
- Test time is also linear (in the length of the test document).
- Thus: Naive Bayes is linear in the size of the training set (training) and the test document (testing). This is optimal.

6.5.5 Derivation of Naive Bayes

Basics

- find the class that is most likely given a document d of text

$$c^* = \arg\max_{c \in \mathcal{C}} \mathbb{P}(c|d) \overset{\text{Bayes}}{=} \arg\max_{c \in \mathcal{C}} \frac{\mathbb{P}(d|c) \cdot \mathbb{P}(c)}{\mathbb{P}(d)} \tag{6.13}$$

- since $\mathbb{P}(x)$ is constant for all classes it can be dropped:

$$c^* = \arg\max_{c \in \mathcal{C}} \mathbb{P}(d|c) \cdot \mathbb{P}(c) = \arg\max_{c \in \mathcal{C}} \mathbb{P}(\langle t_1, \ldots, t_k \ldots t_{n_d} \rangle | c) \cdot \mathbb{P}(c) \tag{6.14}$$

- extreme amount of parameters $\mathbb{P}(\langle t_1, \ldots, t_k \ldots t_{n_d} \rangle | c)$ (one for each combination of a class and a sequence of words)
  - to estimate this, we would need tons of data, which we usually do not have $\Rightarrow$ problem of **data sparseness**

- **Naive Bayes conditional independence assumption** to reduce number of parameters to manageable size:

$$\mathbb{P}(d|c) = \mathbb{P}(\langle t_1, \dots, t_k \dots t_{n_d} \rangle | c) = \prod_{1 \leq k \leq n_d} \mathbb{P}(X_k = t_k | c) \tag{6.15}$$

  where

  - X_k: k'th term of the document

- further simplifying assumption for practical estimation: **positional independence**
  - probability of a term is identical for all positions: $\widehat{\mathbb{P}}(X_{k_1} = t|c) = \widehat{\mathbb{P}}(X_{k_2} = t|c)$
  - note that this assumption is made only for computing the practical estimates!

6.5.6 Features & Language Model

- NB can use any sort of features instead of words
- if we use only words and all words in the corpus, then NB is similar to **language modeling**
  - essentially we have a **unigram** language model for each class
  - we can compute the probabilities of sentences assuming they are in a given class

6.5.7 Violation of Assumptions

- conditional independence is usually **badly violated**: e.g. probability of "Bieber" coming after "Justin" a lot higher than after "Angela"
- positional independence is usually **badly violated** as well: e.g. "Hello" appears a lot more likely at the start of a sentence / doc
- NB is horrible at correctly estimating probabilities
- BUT: classification is about predicting the correct class and NOT correctly estimating probabilities

⇒ NB's estimates are good enough to get good class predictions

6.5.8 Positive Aspects

- robust to nonrelevant features (compared to some more complicated learning methods)
- robust to concept drift (changing of definition of class over time)
- better than methods like trees when there exist many equally important features
- good baseline for text classification (not the best, e.g. SVM is usually better)
- **optimal** if independence assumptions hold (true for some domains)
- very fast
- low storage requirements

6.6 Multinomial Logistic Regression

- both Naive Bayes and kNN are **generative classifiers**
  - used **prior** and **conditional** probabilities to **generate the probability** we are actually interested in (indirect computation)
  - more formally: they learn **joint probability distributions** $\mathbb{P}(\mathcal{X}, \mathcal{C}) = \mathbb{P}(\mathcal{X}|\mathcal{C}) \cdot \mathbb{P}(\mathcal{C})$ and transform them into the actual conditional distribution $\mathbb{P}(\mathcal{C}|\mathcal{X})$ we are interested in (using Bayes)
  - BUT: $\mathbb{P}(\mathcal{X}, \mathcal{C})$ allows to **generate** labeled examples based on their probability occurring
- **discriminative classifiers** learn $\mathbb{P}(\mathcal{C}|\mathcal{X})$ **directly** and use it to classify examples
  - e.g. logistic regression!

6.6.1 Approach

- assume d binary features/ feature indicator functions $f_i : \mathcal{C} \times \mathcal{X} \to \{0, 1\}$ (f_i gives feature i)
- for a given example x, return class c^* according to

$$c^* = \arg\max_{c \in \mathcal{C}} \mathbb{P}(c|x) \tag{6.16}$$

- where

$$\mathbb{P}(c|x) = \frac{\exp\left(\sum\limits_{i=1}^{d} w_i \cdot f_i(c, x)\right)}{\underbrace{\sum\limits_{c' \in \mathcal{C}} \exp\left(\sum\limits_{i=1}^{d} w_i \cdot f_i(c', x)\right)}_{\text{scaling factor}}} \tag{6.17}$$

  where $w = (w_1, \ldots w_d)$ is the **weight vector** we want to learn
  - $\exp(\cdot)$ required to get rid of negative values
  - scaling factor is required to generate outputs between 0 and 1
- simplification:
  - if we are only interested in the ranking of the classes (but NOT the probabilities), we can ignore both $\exp(\cdot)$ and scaling

Learning the Weight Vector

- find weights $w^* \in \mathbb{R}^d$ s.t. the give training data is maximally likely
  - **maximum likelihood estimation**

$$
\begin{aligned}
w^* &= \arg\max_w \prod_{(x,c)\in\mathcal{D}} \mathbb{P}(c|x) \\
&= \arg\max_w \sum_{(x,c)\in\mathcal{D}} \log\left(\mathbb{P}(c|x)\right) \\
&= \arg\max_w \underbrace{\sum_{(x,c)\in\mathcal{D}} \log\left(\frac{\exp\left(\sum_{i=1}^{d} w_i \cdot f_i(c,x)\right)}{\sum_{c'\in\mathcal{C}} \exp\left(\sum_{i=1}^{d} w_i \cdot f_i(c',x)\right)}\right)}_{\log(\mathcal{L}(w))}
\end{aligned}
\tag{6.18}
$$

- $\log(\mathcal{L}(w))$: log-likelihood function
  - convex function $\Rightarrow$ can be efficiently maximized / minimized using **gradient ascent / descent**
  - basic learning idea:
    1) initialize w_0 randomly
    2) until convergence iterate: $w_{t+1} = w_t + \beta\nabla\mathcal{L}(w)$

6.6.2 Overfitting and Regularization

- **overfitting**: weights are optimized too much towards to training data and thus fail to generalize

$\Rightarrow$ **regularization**: adapting objective function in order to penalize "model complexity"

$$w^* = \arg\max_w \sum_{(x,c)\in\mathcal{D}} \log\left(\mathbb{P}(c|x)\right) - \alpha \cdot R(w) \tag{6.19}$$

where

- $R(w)$: regularization function
- α: regularization weight

L_2 Regularization / Ridge Regression

- definition

$$R(w) = L_2(w) = \sum_k w_k^2 \tag{6.20}$$

- easy to to use due to simple gradient (i.e. easy to optimize)
- prefers **small weights**
- can be interpreted as Bayesian classification with Gaussian prior
  - $\alpha \cdot R(w)$ is essentially a Gaussian prior with special parameters

L_1 Regularization / Lasso

- definition
$$R(w) = L_1(w) = \sum_k |w_k| \tag{6.21}$$
- not as easy to use as L_2 due to more complicated gradient
- prefers **sparse weight vectors**

6.7 Beyond Binary Classification

6.7.1 Multivalue Classification

- multilabel classification
- simple solution: **binary relevance learning**
    - for each label/class learn a binary classifier, telling if the example belongs to the class or not
    - at query time, assign example to all classes for which the corresponding classifier outputs *true*

6.7.2 Multinomial Classification

- (multi-class classification)
- simple solution: **one-vs-rest decomposition**
    - train one classifier for each class, predicting a probability if an example belongs to the class or not
    - return class whose classifier gives the highest probability for the example

6.8 Evaluation

6.8.1 Precision & Recall (Measure)

- **precision**: fraction of correct decisions over decisions made
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{6.22}$$
- **recall**: fraction of correct decisions over all data
$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{6.23}$$

6.8.2 F-Measure

- combines recall and precision
$$F = \frac{(1+\beta^2)\cdot P \cdot R}{\beta^2 \cdot P + R} \tag{6.24}$$
where $\beta^2 \in [0, \infty]$
- for $\beta = 1$: **balanced F**, i.e. harmonic mean of P and R (equally weights P and R):
$$F_1 = \frac{2 \cdot P \cdot R}{P + R} \tag{6.25}$$
- $\beta < 1$: emphasizes precision
- $\beta > 1$: emphasizes recall
- harmonic mean is kind of a smooth minimum of P and R
    - minimum punishes really bad performance on either P or R
    - BUT is not smooth and hard to weight
    - e.g. arithmetic mean := 0.5 when returning NO results (too high)

6.8.3 For Multiple Classes

- Micro- vs. Macro Averaging Measure (F_1 measure)
    - **Macro Averaging**
        - compute F_1 for each class
        - compute average of these F_1 values
    - **Micro Averaging**
        - compute TP, FP, FN for each class
        - create sums of TP, FP, FN over all classes
        - compute F_1 for aggregate values
    - note: micro and macro-averaging are identical for precision and recall

6.8.4 N-Fold Cross-Validation

- can compute measure on n-fold cross-validation
- split dataset into n equally sized parts and in each run
    - train on all but one part
    - evaluate on remaining part (validation part)
- cycle validation part through
- aggregate results for each validation part to a final result

6.9 Example Task: Sentiment Analysis

6.9.1 Applications

- reviews: positive or negative?
- products: what do people think about it?
- public sentiment: high consumer confidence?
- politics: what do people think of candidate X?
- prediction: predict outcomes of e.g. elections based on sentiment

6.9.2 Definition

- **detection of attitudes**, i.e. of enduring, affectively colored beliefs, dispositions towards objects or persons
- wording
  - **holder** / **source**: source (e.g. human) showing attitude
  - **target** / **aspect**: target towards the attitude is shown
  - **type** of attitude
    - from a set of types (e.g. like, love, hate, value, desire)
    - or simple weighted **polarity**: weighted values for positive, neutral and negative
  - **text** containing the attitude
- simplest task: **sentiment polarity detection**: is the attitude of given text positive or negative (we focus on this)

6.9.3 Baseline Algorithm

assume we have a training dataset of texts (e.g. reviews) and binary rating (like or dislike)

1) tokenization of texts
2) feature extraction on tokenized texts
3) learn a binary classifier based on the extracted features and the ratings (from the training data)
   - Naive Bayes
   - SVM
   - Decision Trees
   - etc.

Tokenization Issues

- deal with markup
- capitalization (angry people write in CAPS!!!11)
- phone numbers, dates, etc.
- emoticons
- negation
  - add "NOT_" to any word between occurrence of negation and next punctuation:
  - e.g.: "I didn't like the movie, but" → "I didn't NOT_like NOT_this NOT_movie, but"

Feature Extraction

- use words as features
  - only adjectives
  - all words (usually better)
- word occurrence does matter more than frequency (in sentiment analysis!)
  - we can work with binary features for words

Challenges

- text can be tricky (people rarely are precise and concise)
- failed expectations
  - e.g. "This film should be good, as it has all of these great actors. However it sucks."

CHAPTER 7

Sequence Labeling & Part of Speech Tagging

7.1 Introduction

- **learning sequences** is required in many places in NLP
  - POS tagging: annotate each word in a sentence with its syntactic category

7.2 Hidden Markov Models

7.2.1 Markov Chains

Definition

Markov Chain $C = (Q, q_0, q_F, A)$

- set of **states** $Q = \{q_1, \ldots, q_N\}$
- **start** q_0 and **end** state q_F
  - instead of start state, we can have an **initial state distribution** π
  - we might not have a final state at all
- **transition probability matrix** A $((N+2) \times (N+2))$
  - with $\forall i : \sum_j a_{i,j} = 1$
  - $a_{i,j}$: probability of moving from state i to j

Assumptions

- Markov assumption:

$$\mathbb{P}(q_i | q_1, \ldots, q_{i-1}) = \mathbb{P}(q_i | q_{i-1}) \tag{7.1}$$

Computation Example

- compute state sequence probability: e.g. states 3-3-3-3

$$\mathbb{P}(3-3-3-3) = a_{0,3} \cdot a_{3,3} \cdot a_{3,3} \cdot a_{3,3} \tag{7.2}$$

7.2.2 Hidden Markov Model

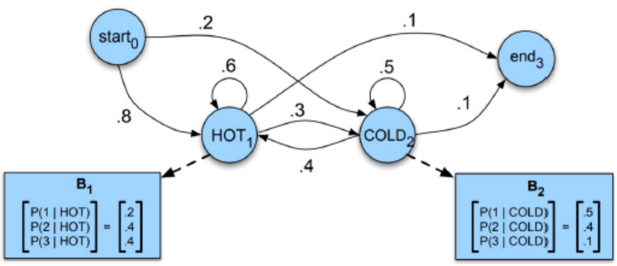
Idea

- Markov Chains: state = output symbol/event
- Hidden Markov models: extension of Markov Chains in which the events/output symbols differ from states
  - states: hidden (e.g. POS tags)
  - events/output symbols: observable (e.g. words for which we search a POS tag)

Definition

Markov Chain $C = (Q, q_0, q_F, A, V, B)$

- set of **states** $Q = \{q_1, \dots, q_N\}$
- **start** q_0 and **end** state q_F
  - instead of start state, we can have an **initial state distribution** π
  - we might not have a final state at all
- **transition probability matrix** A $((N+2) \times (N+2))$
  - with $\forall i : \sum_j a_{i,j} = 1$
  - $a_{i,j}$: probability of moving from state i to j[a]
- **vocabulary** V: set of output symbols
- **emission** probability matrix B with $b_i(k)$: probability of outputting $o_k \in V$ in state $i \in Q$



Assumptions

- Markov assumption:
$$\mathbb{P}(q_i|q_1, \dots, q_{i-1}) = \mathbb{P}(q_i|q_{i-1}) \tag{7.3}$$
- Output independence: output probability of a symbol depends only on the current state
$$\mathbb{P}(o_t|(q_1, \dots, q_t), (o_1, \dots, o_{t-1})) = P(o_t|q_t) \tag{7.4}$$

Problems and Tasks

1) **Observation Sequence Likelihood**: given a HMM $\lambda = (A, B)$ and an observation sequences O, determine the likelihood (i.e. probability) of the sequence $\mathbb{P}(O|\lambda)$
   - **Forward algorithm**
2) **Decoding Observation Sequences**: given a HMM $\lambda = (A, B)$ and an observation sequences O, determine the sequence of hidden states which most likely produced O
   - **Viterbi algorithm**
3) **Learning an HMM**: given an observation sequence O and a HMM without A and B, learn the HMM parameter A and B
   - **Forward-Backward algorithm**

7.3 Observation Sequence Likelihood Estimation

7.3.1 Problem

- given a HMM $\lambda = (A, B)$ and an observation sequences $O = (o_1, \dots, o_T)$, determine the likelihood (i.e. probability) of the sequence $\mathbb{P}(O|\lambda)$

7.3.2 Idea

- different sequences of states can possibly produce a given observation sequence

$\Rightarrow$ sum over all possible state sequences which could produce sequence

$\Rightarrow$ sum over joint probability distribution of given observation sequence and all possible state sequences Q
$$\mathbb{P}(O) = \sum_Q \mathbb{P}(O, Q) = \sum_Q \mathbb{P}(O|Q) \cdot \mathbb{P}(Q) \tag{7.5}$$
  - where the $\mathbb{P}(O, Q)$ is the joint probability distribution defined as
$$\mathbb{P}(O, Q) = \mathbb{P}(O|Q) \cdot \mathbb{P}(Q) = \prod_{t=1}^{T} \mathbb{P}(o_t|q_t) \cdot \prod_{t=1}^{T} \mathbb{P}(q_t|q_{t-1}) \tag{7.6}$$

- **problem**: exponential number of possible state sequences: N^T

7.3.3 Forward Algorithm

- dynamic programming approach (runtime $O(N^2 \cdot T)$)
- fold paths into so called **foward trellis** α ($T \times (N+1)$ matrix)
- $\alpha_t(j)$: probability of being in state j after seeing the first t observations
- **approach**

  1) initialization: $\forall 1 \leq j \leq N$
  $$\alpha_1(j) = a_{0,j} \cdot b_j(o_1) \tag{7.7}$$

  2) recursion (states 0 and F have not outputs): $\forall 1 \leq j \leq N, 1 < t \leq T$
  $$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) \cdot a_{i,j} \cdot b_j(o_t) \tag{7.8}$$

  3) termination
  $$\mathbb{P}(O) = \alpha_T(F) = \sum_{i=1}^{N} \alpha_T(i) \cdot a_{i,F} \tag{7.9}$$

7.4 Decoding Observation Sequences

7.4.1 Problem

- given an HMM $\lambda = (A, B)$ and an observation sequences $O = (o_1, \ldots, o_T)$, determine the sequence of hidden states $Q = (q_1, \ldots, q_T)$ which most likely produced O

7.4.2 Idea

- simple approach

  1) get all possible sequences Q

  2) compute
  $$\arg\max_Q \mathbb{P}(O|Q) \tag{7.10}$$

- **problem**: N^T possible state sequences!
- **solution**: find the **most likely sequence of states directly** using **dynamic programming**

7.4.3 Viterbi Algorithm

- dynamic programming approach (runtime $O(N^2 \cdot T)$)
- computes **viterbi path probabilities** v_t for each timestep t (and according backpointers bp_t for reverse-construction of the state sequence)
- $v_t(j)$: probability of the most likely path producing O until timestep t with $q_t = j$
- **approach**
  1) initialization: $\forall 1 \leq j \leq N$

$$\begin{aligned} v_1(j) &= a_{0,j} \cdot b_j(o_1) \\ bp_1(j) &= 0 \end{aligned} \tag{7.11}$$

  2) recursion: $\forall 1 \leq j \leq N, 1 < t \leq T$

$$\begin{aligned} v_t(j) &= \max_{1 \leq i \leq N} v_{t-1}(i) \cdot a_{i,j} \cdot b_j(o_t) \\ bp_t(j) &= \arg\max_{1 \leq i \leq N} v_{t-1}(i) \cdot a_{i,j} \end{aligned} \tag{7.12}$$

  3) termination:

$$\begin{aligned} v_T(F) &= \max_{1 \leq i \leq N} v_T(i) \cdot a_{i,F} \\ bp_T(F) &= \arg\max_{1 \leq i \leq N} v_T(i) \cdot a_{i,F} \end{aligned} \tag{7.13}$$

- interpretation
  - $v_T(F)$: probability of "best" fitting state sequence
  - $bp_T(F)$: last state of "best" fitting state sequence

Construction of state sequence

1) start with $q_T = bp_T(F)$ (last state)
2) last but one state is $q_{T-1} = bp_{T-1}(q_T)$
3) last but two state is $q_{T-2} = bp_{T-2}(q_{T-1})$
4) etc.

7.5 Learning an HMM

7.5.1 Problem

- given an observation sequence $O = (o_1, \ldots, o_T)$ and the set of states Q, learn transition probability matrix A and emission probability matrix B

7.5.2 Idea: Forward-Backward / Baum-Welch Algorithm

- start with initial estimate (e.g. random)
- iterative process powered by one **main idea**
  1) estimate probabilities by computing the forward probability of an observation and dividing this probability mass along all paths which contributed to the forward probability
- two main quantities involved
  - β: **backward probability**
  - ξ **condition transition probability**

7.5.3 Estimating Transition Probabilities

- intuition:

$$\widehat{a}_{i,j} = \frac{\text{expected number of transitions from state } i \text{ to } j \text{ given the observation sequence}}{\text{expected number of transitions from state } i \text{ given the observation sequence}} \tag{7.14}$$

  - let $\xi_t(i,j)$ probability to transition from state i to j at timestep t
  - numerator: compute $\xi_t(i,j)$ for all timesteps and sum over all timesteps
  - denominator: compute $\xi_t(i,j)$ for each timestep and sum over timesteps and all states (as targets)
- definition:

$$\widehat{a}_{i,j} = \frac{\sum\limits_{t=1}^{T} \xi_t(i,j)}{\sum\limits_{t=1}^{T} \sum\limits_{k=1}^{N} \xi_t(i,k)} \tag{7.15}$$

Computation of ξ

- formal definition

$$\xi_t(i,j) = \mathbb{P}(q_t = i, q_{t+1} = j | O) = \frac{\overbrace{\mathbb{P}(q_t = i, q_{t+1} = j, O)}^{\xi_t^*(i,j)}}{\underbrace{\mathbb{P}(O)}_{\alpha_T(F)}} \tag{7.16}$$

- interpretation of $\xi_t^*(i,j)$
  - situation: we have seen the first t observations and want to be in state i, then transition to state j and want to see the remaining states of the observations
  - recall: **forward probability** $\alpha_t(i)$: probability of being in state i after seeing the first t observations
  - **backward probability** $\beta_t(i)$: probability of seeing the future observations $(o_{t+1}, \ldots, o_T)$ given we are in state i at timestep t (computed similar to $\alpha_t(i)$)

$\Rightarrow$ computation of $\xi_t^*(i,j)$

$$\xi_t^*(i,j) = \mathbb{P}(q_t = i, q_{t+1} = j, O) = \alpha_t(i) \cdot [a_{i,j} \cdot b_j(o_{t+1})] \cdot \beta_{t+1}(j) \tag{7.17}$$

- all together

$$\xi_t(i,j) = \mathbb{P}(q_t = i, q_{t+1} = j | O) = \frac{\overbrace{\mathbb{P}(q_t = i, q_{t+1} = j, O)}^{\xi_t^*(i,j)}}{\underbrace{\mathbb{P}(O)}_{\alpha_T(F)}} = \frac{\alpha_t(i) \cdot [a_{i,j} \cdot b_j(o_{t+1})] \cdot \beta_{t+1}(j)}{\alpha_T(F)} \tag{7.18}$$

7.5.4 Estimating Emission Probabilities

- intuition:

$$\widehat{b}_j(v_k) = \frac{\text{expected number of times in state } j \text{ seeing symbol } v_k}{\text{expected number of times in state } j} \tag{7.19}$$

  - let $\gamma_t(j)$: probability of being in state j at timestep t given the observation sequence
  - numerator: sum $\gamma_t(j)$ over all timesteps where we see symbol v_k
  - denominator: sum $\gamma_t(j)$ over all timesteps

- definition:

$$\widehat{b}_j(v_k) = \frac{\sum\limits_{1 \le t \le T}^{o_t = v_k} \gamma_t(j)}{\sum\limits_{t=1}^{T} \gamma_t(j)} \tag{7.20}$$

Computation of $\gamma_j(v_k)$

- formal definition:

$$\gamma_t(j) = \mathbb{P}(q_t = j | O) = \frac{\mathbb{P}(q_t = j, O)}{\underbrace{\mathbb{P}(O)}_{\alpha_T(F)}} = \frac{\alpha_t(j) \cdot \beta_t(j)}{\alpha_T(F)} \tag{7.21}$$

7.5.5 Complete Algorithm

function FORWARD-BACKWARD(*observations* of len T, *output vocabulary* V, *hidden state set* Q) **returns** $HMM=(A,B)$

initialize A and B
iterate until convergence
E-step

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\alpha_T(q_F)} \ \forall\, t \text{ and } j$$

$$\xi_t(i,j) = \frac{\alpha_t(i)\,a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(q_F)} \ \forall\, t,\ i, \text{ and } j$$

M-step

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1}\sum_{k=1}^{N} \xi_t(i,k)}$$

$$\hat{b}_j(v_k) = \frac{\sum_{t=1 s.t.\, O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

return A, B

Figure 9.16 The forward-backward algorithm.

- part of each E-step: run both backward and forward algorithm to compute β and α values
- parameter-free
- initialization of A and B affects convergence
- initial A and B depend on application
  - speech recognition
    - A is known
    - only need to compute B

7.5.6 Excursion: Computation of β

- Probability of seeing future observations given current state

$$\beta_t(i) = P(o_{t+1} \ldots o_T | q_t = i, \lambda) \tag{7}$$

- Implementation in three steps
  1. Initialization

$$\beta_T(i) = a_{iF} \ 1 \leq i \leq N \tag{8}$$

  2. Recursion Why are states 0 and F not included?

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1})\beta_{t+1}(j) \ 1 \leq i \leq N, 1 \leq t < T \tag{9}$$

  3. Termination

$$\beta_0(q_0) = \sum_{j=1}^{N} a_{0j} b_j(o_1)\beta_1(j) \tag{10}$$

7.6 Part of Speech (POS) Tagging

7.6.1 Problem and Goal

- definition **POS tag**: category to which a word is assigned in accordance with its syntactic functions
  - also known as **word classes**, **syntactic categories**
- annotate each word in a sentence with its syntactic category
- useful for subsequent syntactic parsing

7.6.2 Challenges

- magnitude of the set of tags
  - different tag sets per corpus
  - different tag sets per language (some languages do not have articles...)
- unknown words
- use of sequence information

7.6.3 POS-Tag Classes

two main classes

1) **open classes**
   - nouns
   - verbs
   - adjectives
   - adverbs
2) **closed classes**
   - prepositions (on, under, over)
   - articles (a,an,the)
   - pronouns (she, who, I, others)
   - conjunctions (and, but, or)q
   - auxiliary verbs (can, may, should)
   - particles (up, down, on, off, in, out)
   - numerals (one, two, three)

- not all languages have all subclasses

Nouns

- part of speech inflected for case, signifying a concrete or abstract entity
- two subclasses
  - **proper nouns**: names for particular entities (Germany, Alex, Tanja)
  - **common nouns**: name general items/people (not specific ones) (country, man, woman)

7.7 Learning POS Taggers

- commonly learning of sequences (HMM: hidden states = tags, observations = words)
- training data = annotated corpora
- learning assumes tokenization has taken place (shouldn't → should + n't)
- POS tagging sometimes is considered a solved problem
  - overall accuracy (per token): 0.92 – 0.97 (pretty good)
  - even better if we assume that even judges disagree and some terms are ambiguous
  - BUT: accuracy per sentence (getting a whole sentence right gives one point): ≈ 0.55 (not so good anymore!)

7.7.1 Model

- hidden state are POS tags t_j and **known** (annotated corpus, e.g. Penn Treebank)
- observations are words w_j

Learning Task

1) **learn transition probabilities** A via counting

$$\mathbb{P}(t_i|t_{i-1}) = \frac{c(t_{i-1}, t_i)}{c(t_{i-1})} \tag{7.22}$$

   where

   - $c(t_{i-1}, t_i)$: number of times tag t_i follows tag t_{i-1}
   - $c(t_{i-1})$: number of times tag t_{i-1} is used

2) **learn emission probabilities** B via counting

$$\mathbb{P}(w_i|t_i) = \frac{c(t_i, w_i)}{c(t_i)} \tag{7.23}$$

   where

   - $c(t_i, w_i)$: number of times tag t_i was used on word w_i
   - $c(t_i)$: number of times tag t_i was used

Prediction Task

- given the learned HMM, we want to **decode the observation sequence** (i.e. find the sequence of best POS tags)

$\Rightarrow$ Viterbi algorithm

7.7.2 Considerations

- main assumption:
$$\mathbb{P}(t_1^n) = \prod_{i=1}^{n} \mathbb{P}(t_i|t_{i-1}) \tag{7.24}$$
- practical applications use more history, e.g. **trigrams**
$$\mathbb{P}(t_1^n) = \prod_{i=1}^{n} \mathbb{P}(t_i|t_{i-1}, t_{i-2}) \tag{7.25}$$
- **problem**: cannot be handled by standard Viterbi algorithm
- **solution**: adapt algorithm to find
$$\arg\max_{t_1^n} \left[\prod_{i=1}^{n} \mathbb{P}(w_i|t_i) \cdot \mathbb{P}(t_i|t_{i-1}, t_{i-2}) \right] \cdot \mathbb{P}(t_{n+1}|t_n) \tag{7.26}$$
  - add special symbols t_{-1}, t_0, t_{n+1}
- another **problem**: **data sparsity**
  - counting as before yields unreliable data
  - solution: **deleted interpolation**
$$\mathbb{P}_{di}(t_i|t_{i-1}, t_{i-2}) = \lambda_3 \cdot \mathbb{P}(t_i|t_{i-1}, t_{i-2}) + \lambda_2 \cdot \mathbb{P}(t_i|t_{i-1}) + \lambda_1 \cdot \mathbb{P}(t_i) \tag{7.27}$$
    where $\sum_i \lambda_i = 1$
- one more **problem**: **unknown words**
  - use morphological clues

CHAPTER 8

Grammar and Parsing

8.1 Fundamentals

- **constituency**
  - detect groups of words which behave as a single unit, called **constituents**
  - inventory of constituents is core of grammar development

8.2 Context Free Grammars (CFG)

- other name: **phrase-structure** grammar
- equivalent to **Backus-Naur form**
- CFG for natural language
  - idea: grammar on constituents

8.2.1 Formal Model

CFG $G = (N, \Sigma, R, S)$ where

- N: set of **non-terminal** symbols
- Σ: set of **terminal** symbols ($\Sigma \cap N = \emptyset$)
- R: set of rules/productions, each of the form $A \to \beta$ where
  - $A \in N$, i.e. A is a non-terminal
  - $\beta \in (\Sigma \cup N)^*$, i.e. β is a string of symbols
- $S \in N$: start symbol

Language of a CFG

- language of CFG G: $\mathcal{L}(G) = \{w | w \in \Sigma^* \wedge S \overset{*}{\Rightarrow} w\}$
  - set of **non-terminal-free strings** which can be derived starting at S
- **direct derivation**: $\alpha A \gamma \Rightarrow \alpha\beta\gamma$
  - if $A \to \beta \in R$, then $\alpha A \gamma$ directly derives $\alpha\beta\gamma$
- **derivation**: $\alpha_1 \overset{*}{\Rightarrow} \alpha_m$
  - if there exists $\alpha_i \in (\Sigma \cup N)^*$ with $\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_m$, then α_1 derives α_m
- **grammatical sentences**: sentences derived from S
- **ungrammatical sentences**: all other

8.2.2 Equivalence

- two distinct grammars G_1 and G_2 can generate the same language
- G_1 and G_2 are **strongly equivalent** if
  1) they accept the same languages AND
  2) they assign the same parse tree to every sentence
- G_1 and G_2 are **weakly equivalent** if
  1) they accept the same languages AND
  2) do NOT assign the same parse tree to every sentence

8.2.3 Chomsky Normal Form (CNF)

- grammar G is in CNF if and only if the following conditions hold
  1) each rule in R is in one of the following forms
     - $A \to a$ $(a \in \Sigma)$OR
     - $A \to BC$
  2) it is ϵ-free
- idea: **binary branching** (no parse tree will have more than two branches at any level)
- usage: allows for efficient syntactic parsing using **CKY algorithm**
- every grammar can be transformed into CNF by splitting rules

Modeling NL

- terminals := words
- non-terminal := abstractions over terminals
- CFGs can be used for **language generation** and **structure assignment**

8.2.4 Derivation Representations

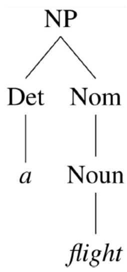
Bracketed Notation

Bracketed notation

$[_S\ [_{NP}\ [_{Pro}\ \text{I}]]\ [_{VP}\ [_V\ \text{prefer}]\ [_{NP}\ [_{Det}\ \text{a}]\ [_{Nom}\ [_N\ \text{morning}]\ [_{Nom}\ [_N\ \text{flight}]]]]]]$

Parse Tree

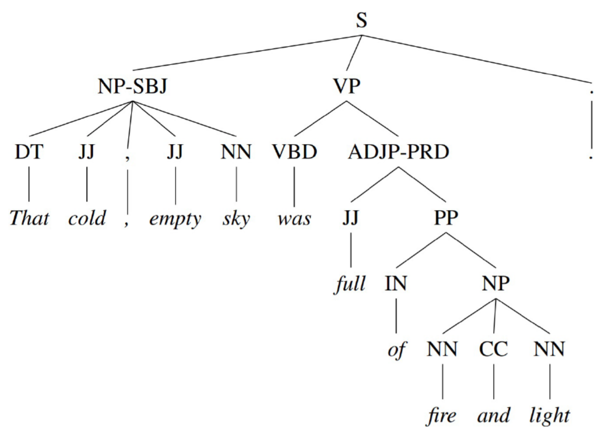
- node is said to **dominate** all nodes in tree
- **syntactic parsing**: maps a word to its parse tree



8.3 Treebanks

- **treebank**: repository of parsed sentences for a given language
- main assumption: sufficiently complex grammar allows parsing every grammatical sentence of a given language
- well known example: **Penn Treebank**
- note: rules for parsing can be directly extracted from treebanks

```
((S
   (NP-SBJ (DT That)
     (JJ cold) (, ,)
     (JJ empty) (NN sky) )
   (VP (VBD was)
     (ADJP-PRD (JJ full)
       (PP (IN of)
         (NP (NN fire)
           (CC and)
           (NN light) ))))
   (. .) ))
```



8.4 Lexicalized Grammars

- problems of CFGs: **focus on rules**
  - poor context modeling
  - long, redundant set of rules
- **lexicalized grammars** focus on the **lexicon**

8.4.1 Components

- set of categories $\mathcal{C}$
- lexicon: $\mathcal{W} \rightarrow 2^{\mathcal{C}}$ (i.e. maps words to a category or combinations of categories)
- set of rules for combining categories

Categories

- two types
  1) set of atomic elements $\mathcal{A} \subseteq \mathcal{C}$
  2) single-argument functions $\mathcal{C} \rightarrow \mathcal{C}$
     - $(X \backslash Y) \in \mathcal{C}$ if $X, Y \in \mathcal{C}$ and
     - $(X/Y) \in \mathcal{C}$ if $X, Y \in \mathcal{C}$
- interpretation $(X \backslash Y)$
  - seeks value of type Y to its **left**
  - returns a value of X
- interpretation (X/Y)
  - seeks value of type Y to its **right**
  - returns a value of X

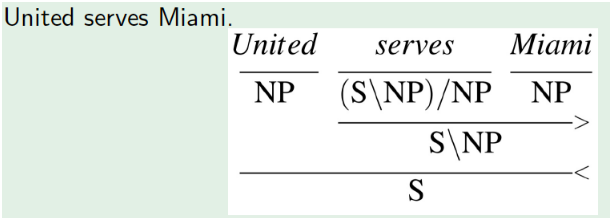
Lexicon

- assignment of categories to words
- nouns are commonly assigned to atomic categories
- verbs can be assigned to composite categories (allows for subcategorization)

- flight: N
- Miami: NP
- cancel: (S\NP)/NP
- give: ((S\NP)/NP)/NP

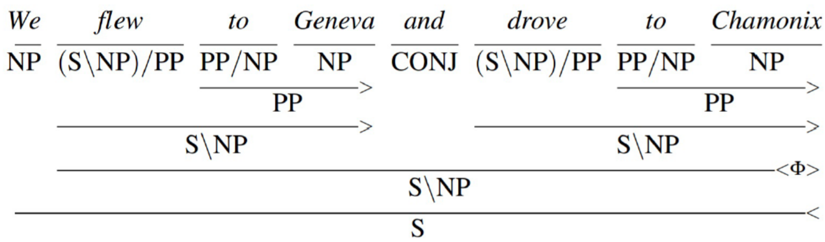
Rules

- two basic templates
  1) **forward function application** $X/Y Y \rightarrow X$
  2) **backward function application** $Y X\backslash Y \rightarrow X$
- output in both cases: value of the function being applied



Metarule

- $X\, CONJUNCTION\, X \rightarrow X$

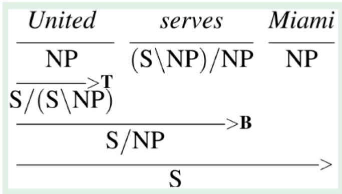


8.4.2 Further Development

- core of **categorical grammar** (most facts in lexicon, only 3 rules BUT **not more expressive than CFG**)
- extended to **Combinatory Categorical Grammar** (CCG) by **operators over functions**

8.4.3 Combinatory Categorical Grammar (CCG)

- additionally: **operators over functions**
- two operators
  - **composition** (denoted as B in rules)
    1) **forward composition**: $X/Y Y/Z \rightarrow X/Z$
    2) **backward composition** $Y\backslash Z X\backslash Y \rightarrow X\backslash Z$
  - **type raising** (denoted as T in rules)
    1) $X \rightarrow T/(T\backslash X)$
    2) $X \rightarrow T\backslash(T/X)$



Properties

- allows for linear processing (**computationally really relevant lexicalized grammar**)
- does not allow conjunction
- allows processing long-distance dependencies

CHAPTER 9

Syntactic and Probabilistic Parsing

9.1 Problem of Ambiguity

- ambiguity is a major problem for parsers
- two types of ambiguity
  - part-of-speech ambiguity (words can have one more possible POS tags) **structural ambiguity**: several parses possible for the same sentence

9.1.1 Structural Ambiguity

- **attachment ambiguity**
  - constituent can be attached at different places in the parse tree
  - e.g.: "We saw the Eiffel Tower flying to Paris."
- **coordination ambiguity**
  - different parts of phrases can be conjoined by a conjunction like "and"
  - "old (men and women)" vs "(old men) and women"
- more general: **syntactic ambiguity**: many grammatically correct but semantically unreasonable parses for naturally occurring sentences
  - parsers need to perform **syntactic disambiguation** in order to find the right parse for a sentence

9.2 Cocke-Kasami-Younger (CKY) Parsing

9.2.1 Idea

- dynamic programming approach
  - exploits **context-freeness**

- if we found a constituent in a part of the input, we can record its presence and use it for any later derivation if required

- assumes grammar is in CNF (Chomsky Normal Form)

9.2.2 Chomsky Normal Form Normalization

process of transforming any (ϵ-free) CFG to CNF

1) copy all conforming rules unchanged
2) for each terminal γ in **mixed rules** (RHS contains terminal and non-terminals) create **dummy terminals** X_γ
   - replace occurrences of γ with X_γ
   - add rules $X_\gamma \rightarrow \gamma$
3) remove all **unit productions** (rules with only a single non-terminal on RHS)
   - assume rule $X \rightarrow Y$
   - search for all rules making use of X and add replace X with Y
   - delete rule $X \rightarrow Y$
4) make all rules binary
   - assume non-binary rule $A \rightarrow BC\gamma$
   - create rules $A \rightarrow X\gamma$ and $X \rightarrow BC$

main advantage of CNF: **binary parse trees** for all derivations (except for leaves)

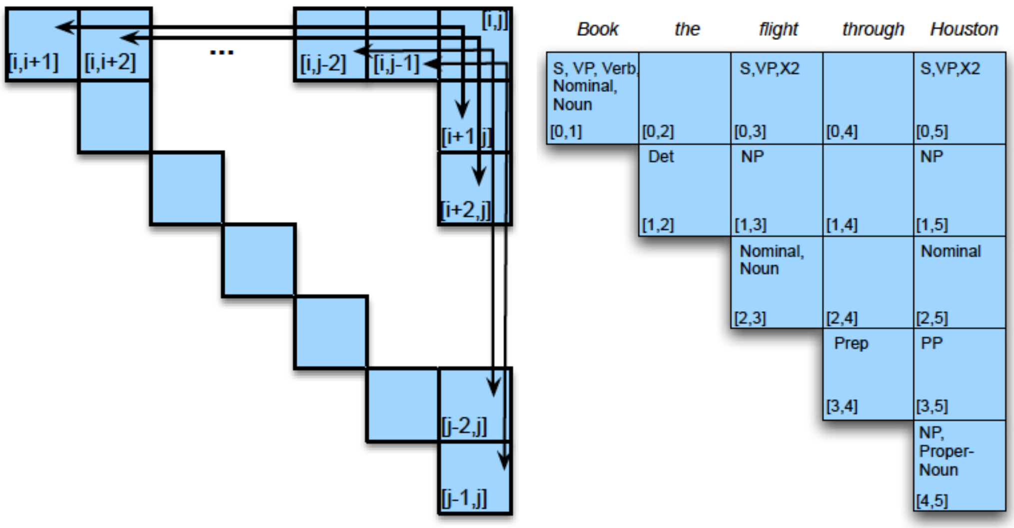
9.2.3 CKY Recognition

Representation

- due to CNF: parse trees are **binary** $\Rightarrow$ two-dimensional matrix can be used to represent the whole tree
- assuming a given sentence with n words, we index the **gaps** between the words (including before first and after last word) from 0 to n
- use upper right triangle of $(n+1) \times (n+1)$ matrix C to encode parse tree
- cell $c_{i,j}$: contains the set of non-terminals that represent all the tokens between positions i and j (in the input)

  $\Rightarrow$ cell $c_{0,n}$ represents entire input

  - since we have binary rules, every non-terminal represented by a cell $c_{i,j}$ can be split into two parts at a position k (in the input) such that $i < k < j$
  - left part of derivation $c_{i,k}$ must lie to the left of entry $c_{i,j}$ in row i
  - right part of derivation $c_{k,j}$ must lie below $c_{i,j}$ in column j



- lower diagonal contains possible constituents for the according words
- recognition algorithm fills this table correctly

Algorithm

Algorithm 1 Recognize(grammar G, sentence s of length n)

for $j \leftarrow 1 \dots n$ **do** ▷ over columns
  for $\{A | A \rightarrow s[j] \in G\}$ **do** ▷ fill lower diagonal
    $table[j-1, j] \leftarrow table[j-1, j] \cup \{A\}$
  end for
  for $i \leftarrow j-2 \dots 0$ **do** ▷ over rows
    for $k \leftarrow i+1 \dots j-1$ **do** ▷ possible split points
      for $\{A | A \rightarrow BC \in G\}$ **do** ▷ all possible rules resulting in splits
        if $B \in table[i,k] \wedge C \in table[k,j]$ **then**
          $table[i,j] \leftarrow table[i,j] \cup \{A\}$
        end if
      end for
    end for
  end for
end for

Recognizer is not a Parser

- it only **recognizes** if an input is part of a grammar or not (if S is in cell $c_{0,n}$ after running, it accepts, otherwise not)
- parsers need to return all possible parses of a sentence!

9.2.4 CKY Parsing

- recognition can be extended to parsing by
  1) pair non-terminals with pointers to entries from which it was derived

2) permit for multiple versions of a non-terminal in each cell

⇒ using this information we can return all possible parses

- problem: possibly exponential many parses
- solution: return only the "best" parse
  - **probabilistic CKY**: probabilistic model + modified Viterbi algorithm

9.3 Probabilistic CKY

9.3.1 Probabilistic CFG (PCFG)

- same as CFG BUT
  - rules are extended, s.t, each rule has a probability: $A \rightarrow \beta[\mathbf{p}]$
  - meaning: $\mathbb{P}(A \rightarrow \beta) = p$ or more precisely $\mathbb{P}(A \rightarrow \beta|A) = p$
- **consistent PCFG**: for all $A \in N$ it holds that

$$\sum_{\beta} \mathbb{P}(A \rightarrow \beta) = 1 \tag{9.1}$$

9.3.2 PCFGs for Disambiguation

- PCFG assigns a probability to each parse tree T of a given sentence $S \Rightarrow$ we can return the one with the highest probability
- definition for parse tree T with n non-terminal nodes (which have to expanded by n rules $LHS \rightarrow RHS$)

$$\mathbb{P}(T, S) = \mathbb{P}(S|T) \cdot \mathbb{P}(T) = \mathbb{P}(T) = \prod_{i=1}^{n} \mathbb{P}(RHS_i|LHS_i) \tag{9.2}$$

- note: since T includes all words in S: $\mathbb{P}(S|T) = 1$

Formalization

- **yield of a parse tree** T: string S corresponding to the tree
- best parse tree:

$$\begin{aligned}
\widehat{T}(S) &= \arg \max_{\{T|yield(T)=S\}} \mathbb{P}(T|S) \\
&\overset{\text{Bayes}}{=} \arg \max_{\{T|yield(T)=S\}} \frac{\mathbb{P}(T, S)}{\mathbb{P}(S)} \\
&= \arg \max_{\{T|yield(T)=S\}} \mathbb{P}(T, S) \\
&= \arg \max_{\{T|yield(T)=S\}} \mathbb{P}(T)
\end{aligned} \tag{9.3}$$

9.3.3 PCFGs for Language Modeling

- PCFGs can describe (long-distance) dependencies which cannot be described by n-gram models
- probability of a sentence can be computed as

$$\mathbb{P}(S) = \sum_{\{T|yield(T)=S\}} \mathbb{P}(T, S) = \sum_{\{T|yield(T)=S\}} \mathbb{P}(T) \tag{9.4}$$

- note: when converting a PCFG to CNF, we also have to adapt the probabilities

9.3.4 Probabilistic CKY Parsing for PCFGs

- representation: tensor of dimensions $(n+1) \times (n+1) \times |N|$ (N: set of non-terminals)
- $c_{i,j,A}$: probability of constituent A spanning the interval i to j (of the input)
- algorithm: very similar to standard CKY, BUT
  - make use of probabilities
  - keep backpointers

```
function PROBABILISTIC-CKY(words,grammar) returns most probable parse
                                                  and its probability
  for j ← from 1 to LENGTH(words) do
     for all { A | A → words[j] ∈ grammar}
        table[j−1, j,A] ← P(A → words[j])
     for i ← from j−2 downto 0 do
        for k ← i+1 to j−1 do
              for all { A | A → BC ∈ grammar,
                             and table[i,k,B] > 0 and table[k,j,C] > 0 }
                    if (table[i,j,A] < P(A → BC) × table[i,k,B] × table[k,j,C]) then
                       table[i,j,A] ← P(A → BC) × table[i,k,B] × table[k,j,C]
                       back[i,j,A] ← {k,B,C}
     return BUILD_TREE(back[1, LENGTH(words), S]), table[1, LENGTH(words), S]
```

9.4 Augmented PCFGs

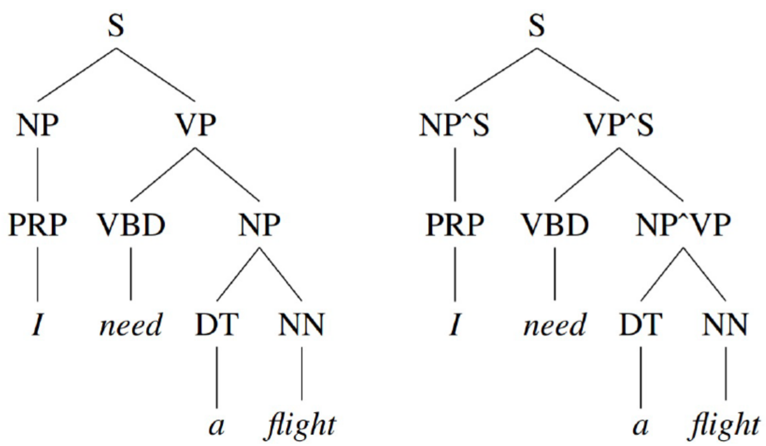
9.4.1 Problems of PCFGs as Probability Estimators

1) **poor independence assumptions**
   - rules modeled as context-independent
   - not correct for many languages (e.g. English)
2) **lack of lexical conditioning**
   - i.e. lack of lexical sensitivity of words in parse tree
   - parse trees independent of syntactic facts about specific words (e.g. some words bind stronger to each other)

- not correct many languages (e.g. English)

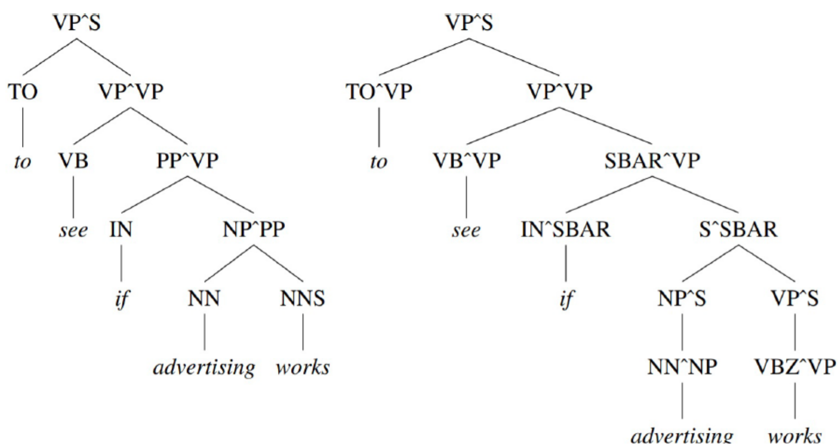
9.4.2 Splitting Non-Pre-Terminal Nodes

- solves problem of poor independence assumption
- idea: make **non-pre-terminals** dependent on parent (**pre-terminal**: node with a terminal as child)
- implementation: annotate them with the parent



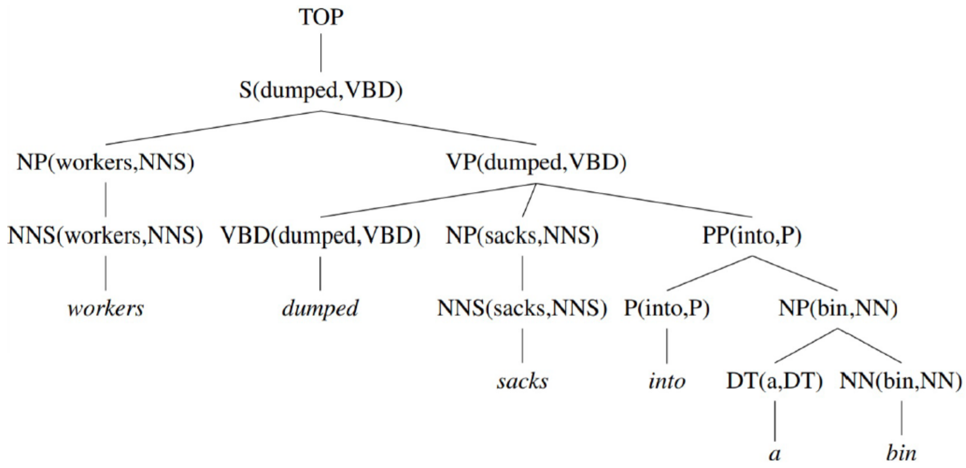
9.4.3 Splitting Pre-Terminal Nodes

- can be done in addition to splitting non-pre-terminal nodes
- solves problem of lexical conditioning
- idea: make pre-terminal nodes (and thus terminal nodes) dependent on their parent
- implementation: annotate with parent



9.4.4 Probabilistic Lexicalized PCFGs

- alternative approach for coping with problems of PCFGS
- instead of modifying the grammar rules, we modify the probabilistic model of the parser to allow for lexicalized rules
  - i.e. modify rules to account for lexicon entries
    - VP(dumped) → VBD(dumped) NP(sacks) PP(into)



Lexicalized Parsing

- make some further independence assumptions to break down each rule so that we would estimate the probability of a rule as the product of smaller independent probability estimates for which we could acquire reasonable counts
- one such parser: **Collins Parser**

Collins Parser

- idea: thinks of RHS of rules as head non-terminal together with the non-terminals left of the head and right of the head

$$LHS \rightarrow L_n \dots L_1 \mathbf{H} R_1 \dots R_m \tag{9.5}$$

- simplification:
  - add stop symbol left and right of the rule
  - compute MLE probabilities for rule: compute MLE probabilities for head, left and right side and aggregate
    - apply **generative model**: first head, then left, then right

dumped sacks into

1. Generate head: P(H|LHS) = P(VBD(dumped,VBD) | VP(dumped,VBD))
2. Generate left dependent: P_l(STOP| VP(dumped,VBD) VBD(dumped,VBD))
3. Generate right dependents
   - P_r(NP(sacks,NNS | VP(dumped,VBD), VBD(dumped,VBD))
   - P_r(PP(into,P) | VP(dumped,VBD), VBD(dumped,VBD))
   - P_r(STOP | VP(dumped,VBD), VBD(dumped,VBD))

9.5 Probabilistic CCG Parsing

9.5.1 Ambiguity in CCGs

- ambiguity in CFGs caused by **rules**
- ambiguity in CCGs caused by **lexicon**
  - large number of complex lexical categories combined with the very general nature of the grammatical rules

9.5.2 Parsing

- option 1: apply CKY - **problems**
  - large number of possible categories added to the table
  - large, but sparse tensor with lots of **zombie constituents**

  ⇒ solution: **supertagging**
    - assign **most probable** lexicon entries to each cell
    - building a supertagger: HMMs
- option 2: A^* Parser - model parsing as **heuristic search problem**
  - **cost function** $f(n)$ composed of
    - exact cost function $g(n)$: exact cost of partial solution
    - heuristic approximation $h(n)$: approximation cost to complete partial solution to a full one
  - require condition (for optimality): $f(n) = g(n) + h(n) \leq f^*(n)$ (i.e. we have to **underestimate** real cost)

A^* Parsing

- assumptions:
  - assume supertagger with probability scores
  - assume rules do not influence model (we only work with the lexicon)
- given a sentence S of length $|S|$ and derivation D with tag sequence T we have
- node $n = (S, T)$: combination of sentence $S = (s_1, \dots s_{|S|})$ and tag sequence $T = (t_1, \dots, t_{|T|})$

$$\mathbb{P}(D, S) = \mathbb{P}(T, S) = \prod_{i=1}^{|S|} \mathbb{P}(t_i | s_i) \tag{9.6}$$

- note: equation above is a **utility**, but we want a **cost measure**
- easier if g is **additive cost measure**

$$g((S, T)) = \sum_{i=1}^{|S|} -\log\left(\mathbb{P}(t_i | s_i)\right) \tag{9.7}$$

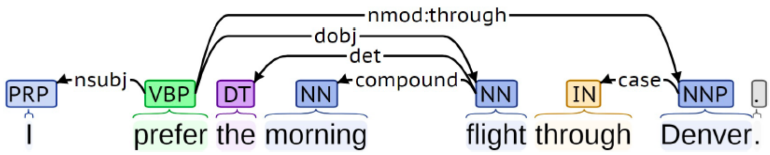
- more general: let n stand for sequence s_i^j with tags t_i^j where $1 \leq i < j \leq |S|$
- heuristic function: worst possible costs from 1 to i and j to n

$$\begin{aligned} g(s_i^j, t_i^j) &= \sum_{k=i}^{j} -\log\left(\mathbb{P}(t_k | s_k)\right) \\ h(s_i^j, t_i^j) &= \sum_{k=1}^{i-1} \max_{t \in \text{tags}} \left(-\log(\mathbb{P}(t | s_k))\right) + \sum_{k=j+1}^{n} \max_{t \in \text{tags}} \left(-\log(\mathbb{P}(t | s_k))\right) \end{aligned} \tag{9.8}$$

- approach:
  - initialize storage with pairs of words (from sentence) and possible tags for each words
  - in each step
    - remove node with minimal h from storage
    - check if it is a complete solution
    - if not generate new nodes based on the rules of the CCG and the node (and add them to the storage)
  - process terminates if either a complete solution is found or storage is empty

9.6 Dependency Parsing

- idea: compute **typed dependency structure**
  - analysis as token level
  - focus not on word order but **dependency relations**
  - typed, because relations are from fixed vocabulary
- goal: make relations hidden in sentence structure explicit (e.g. long distance dependencies)



9.6.1 Formal Specification

- **dependency structure** $G = (V, E, I)$ is graph with
  - V: set of words in the phrase
  - $E \subseteq V^2$: set of directed arcs
  - $I : E \to T$: labeling functions for edges (labels edges with types from set of types T)
- **head**: source of an edge
- **dependent**: target of edge
- **requirements**
  1) **single designated root node** without incoming edges
  2) **each node has exactly one incoming arc** (except for root)
  3) there is a **unique path** from the root node to each node in V

9.6.2 Property: Projectivity

- edge is called **projective** if there exists a path from the head to every word that lies between the head and the dependent in the sentence
- dependency parse tree is **projective** if all of its arcs are projective
- important property
  - most approaches can only produce such trees, i.e. sentences without such a tree, can only be parsed with errors by the parser
  - dependency trees generated from CFG derivation trees are always projective

9.6.3 Dependency Treebanks

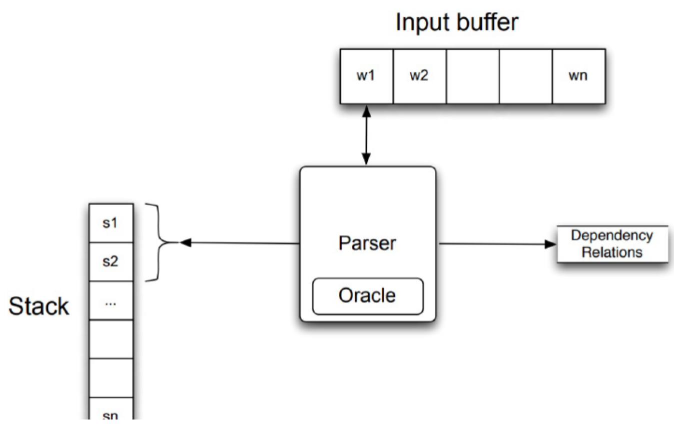
- can be generated from constiutent treebanks
- two main tasks involved in generation
    1) identify head-dependent relations in the structure
    2) identify correct dependency relations for these relations
- usually post-processing by experts to correct for errors

9.6.4 Transition-Based Dependency Parsing

- class of algorithms to compute dependency trees
- very common algorithm: **shift-reduce parsing**
- works with
    - an oracle
    - a stack
    - a list of tokens to be parsed

Shift-Reduce Parsing

- general idea :
    - **shift** input tokens one by one onto stack
    - **reduce**: top two elements of the stack are checked for a head-dependent relation (both ways)
        - oracle checks if such a relation is present
        - if true, the current dependency tree is adapted



- important notion: **configuration** - set of
    - current stack
    - current input buffer
    - current (partial) dependency tree

⇒ algorithm transitions between configurations

- model parsing as **search algorithm** in configuration space

Configuration Search Space

- initial configuration
  - stack = ROOT (EMPTY)
  - input buffer = input sentence
  - relations = empty
- final configuration
  - stack = ROOT (EMPTY)
  - input buffer = empty
  - relations = complete parse (dependency) tree
- **transitions**: **arc standard approach** - 3 standard operations
  - **LeftArc**
    1) assert head-dependent relation between the word at the top of stack and word directly beneath it
    2) remove lower word from stack (as a node can have only one incoming edge)
  - **RightArc**
    1) assert head-dependent relation between the second word of the stack and the top of the stack
    2) remove the top of the stack
  - **Shift**
    1) remove first token from the input buffer and push it onto the stack
- search algorithm:

```
DependencyParse(Words w)
state ← {[Root], [w], []}; initial configuration
while state is not final
do t ← ORACLE(state); //choose a transition operator to apply
    state ← APPLY(t, state); //apply it, creating a new state
return state
```

  - always produces projective trees

Learning the Oracle

- use ML to learn a model predicting a transition given a configuration
- required training data: pairs of configurations and transitions
- problem: treebanks have only complete trees but not the pairs we need
- solution: **generate training data** using **training** oracle
  - simulate shift-reduce algorithm on sentences AND correct dependency tree
  - take transitions given by the **training oracle** (choosing transition based on correct dependency tree)
  - one training example: configuration + chosen transition

Training Oracle

- Choose LEFTARC if it produces a correct head-dependent relation
- Otherwise, choose RIGHTARC if
  1. it produces a correct head-dependent relation given the reference parse
  2. all of the dependents of the word at the top of the stack have already been assigned
- Otherwise, choose SHIFT

CHAPTER 10

Word Vectors

10.1 Vector Model

- model the meaning of a word as a **vector** of features (sometimes called **embedding**)
  - using information about the words it is surrounded by
- intuition behind idea: words have similar meaning if they have similar word contexts
- four models
  - mutual-information weighted word co-occurrence matrices (sparse vectors)
  - singular value decomposition(SVD) and Latent Semantic Analysis (LSA)(dense)
  - neural network inspired models (dense)
  - brown clusters (dense)

10.1.1 Co-Occurrence Matrix (Sparse Vector Model

10.1.2 Term Document Matrix

- assume we have D documents and a vocabulary of size V
- **term document matrix**: matrix M with V rows and D columns
- entry $M_{i,j}$ contains frequency of term i in document d (term frequency)
- document: count vector $d \in \mathbb{N}^V$
- **word**: count vector $w \in \mathbb{N}^D$

$\Rightarrow$ two words are similar if their vectors are similar

- **problem**: documents are poor context models for words

10.1.3 Word-Context Matrix

- also known as word-word matrix
- idea: use smaller context (than a document) in the matrix
- word representation: vector of length V over counts of context words
  - e.g. counts wrt. how often a word appears in a given windows around the term
- word-context matrix: $V \times V$
- **problem**: raw word frequency is not a great measure of association between words
  - very skewed
  - high-frequency words are commonly not very informative

10.1.4 Pointwise Mututal Information (PMI)

- idea: consider whether a context word is **informative** about the target word
  - concept: how much more do events x and y co-occur than fi they were independent

$$PMI(w, w') = \log_2 \left(\frac{\mathbb{P}(w, w')}{\mathbb{P}(w) \cdot \mathbb{P}(w')} \right) \tag{10.1}$$

  where w, w' are words (and w' is usually the context)
- $PMI(w, w') \in [-\infty, \infty]$
- **problem**:negative values
  - things are co-occurring less than expected by chance
  - unreliable without extreme amount of data
  - solution: replace negative values by 0 **PPMI**

$$PPMI(w, w') = \max\left(PMI(w, w'), 0\right) = \max\left(\log_2\left(\frac{\mathbb{P}(w, w')}{\mathbb{P}(w) \cdot \mathbb{P}(w')}\right), 0\right) \tag{10.2}$$

PPMI Computation from Word-Context Matrix

- assume matrix with W words and C contexts
- $f_{i,j}$: number of times word i occurs in context j (entry (i, j) of matrix

$$p_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

$$p_{i*} = \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

$$p_{*j} = \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

$$pmi_{ij} = log_2 \frac{p_{ij}}{p_{i*}p_{*j}}$$

$$ppmi_{ij} = \begin{cases} pmi_{ij} & \text{if } pmi_{ij} > 0 \\ 0 & \text{otherwise} \end{cases}$$

- **problem**: PMI is biased towards **infrequent events**
  - very rare words have very high PMI values
- **solutions**:
  - Laplace smoothing OR
  - give rare words slightly higher probabilities

$$\begin{aligned} PPMI_\alpha(w, w') &= \max\left(\log_2\left(\frac{\mathbb{P}(w, w')}{\mathbb{P}(w)\cdot\mathbb{P}_\alpha(w')}\right), 0\right) \\ \mathbb{P}_\alpha(w) &= \frac{\text{count}(w)^\alpha}{\sum_{w'}\text{count}(w')^\alpha} \end{aligned} \tag{10.3}$$

  - helps since $\mathbb{P}_\alpha(w) > \mathbb{P}(w)$ for rare w

10.1.5 Considerations

- real matrices are often **very sparse** and 50.000×50.000
- size of windows depends on goals
- 2 kinds of co-occurrences
  - **first-order co-occurrence** (syntagmatic association): two words are typically nearby each other
  - **second-order co-occurrence** (paradigmatic association): words with similar neighbors

10.1.6 Problems of Sparse Vector Models

- vectors are **long** but **sparse**

⇒ many weights need to be learned in ML approaches

- explicit counts usually do not generalize too well
- models usually bad in capturing **synonymy**
    - "car" is represented as another context as "automobile"

10.2 Similarity between Vectors

10.2.1 Naive Approach: Dot Product

- dot product between vectors $v, w \in \mathbb{R}^D$

$$sim(v, w) = v \cdot w = \sum_{i=1}^{D} v_i \cdot w_i \tag{10.4}$$

- problem: metric sensitive to word frequency
    - dot product is longer if vector is longer and vectors are longer if they have higher values in each dimension
- solution: **cosine-similarity**

10.2.2 Cosine-Similarity

- normalize dot-product by vector length

$$cossim(v, w) = \frac{v \cdot w}{|v| \cdot |w|} = \frac{\sum_{i=1}^{D} v_i \cdot w_i}{\sqrt{\sum_{i=1}^{D} v_i^2} \cdot \sqrt{\sum_{i=1}^{D} w_i^2}} \tag{10.5}$$

- interpretation: cosine of angle between v and w
- $cos - sim(v, w) \in [-1, +1]$
    - if only non-negative vector entries: $cossim(v, w) \in [0, +1]$

10.2.3 Syntax-Based Similarity

- idea: two words are similar if they have **similar syntactic contexts**
- result: word vector $w \in V \cdot R$ where each entry is the value for a (context,grammatical relation) pair (R: number of grammatical relations)
- can be compressed to a vector of size V by summing up counts for all grammatical relations of a context
  - difference to simple word-context counts: we only count instances of a context which stands in a grammatical relation to the word and not ALL instances

10.3 Counting Based Dense Vector Models

- idea: represent a word by **short** and **dense** representation by **learning** it
- concept: approximate N-dimensional dataset using fewer dimensions (**dimensionality reduction**)
  - rotate axes into a new space
  - order dimensions by how much variance of the original dataset they capture and get rid of low-variance dimensions (e.g. PCA)

10.3.1 Singular Value Decomposition

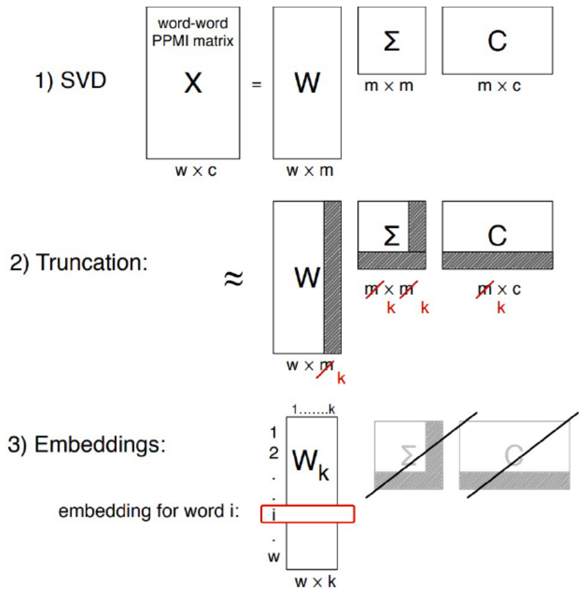
- definition: every rectangular $w \times c$ matrix X equals the product of 3 matrices

$$X = W \cdot S \cdot C \tag{10.6}$$

  where
  - W: matrix of **latent word representations**
    - dimension: $w \times m$
    - columns are ordered by the amount of variance in the dataset each new dimension accounts for
  - S: **singular value matrix**
    - dimension: $m \times m$ diagonal matrix
    - expresses the importance of each dimension
  - C: **context matrix**
    - dimension: $m \times c$
    - columns corresponding to original contexts
    - m rows corresponding to singular values

Latent Semantic Analysis

- SVD applied to word-context matrices
- idea: keep **top-k** dimensions and represent words by **latent representation** in $W \Rightarrow$ result
- in practice: $k \approx 300$



- positive aspects:
  - can be seen as a process for removing noise from data
  - smaller number of dimensions make it easier to use ML approaches
- problem: interpretability is worse

10.4 Prediction Based Dense Vector Models

- idea: learn representations as part of the process of **word prediction**
- examples
  - **skip-grams**: predicts context given input word
  - **CBOW**: predicts a word given a context
- process: train a neural network to predict neighboring words
- advantages:
  - fast, easy to train (faster than SVD)
  - usually pretrained models found online (e.g. word2vec)

10.4.1 Skip-Grams

- skip-grams are given a word and predict a context (word)
- learns two different representations of a word w: the **word embedding** v and the **context embedding** c
    - embeddings are encoded in **word matrix** W and **context matrix** C
    - column i of word matrix is **word embedding** representation of word i
    - row i of input matrix is **context embedding** representation of word i
- since we need only one representation, we can either use one of the two or aggregate the two in some way

Prediction Task

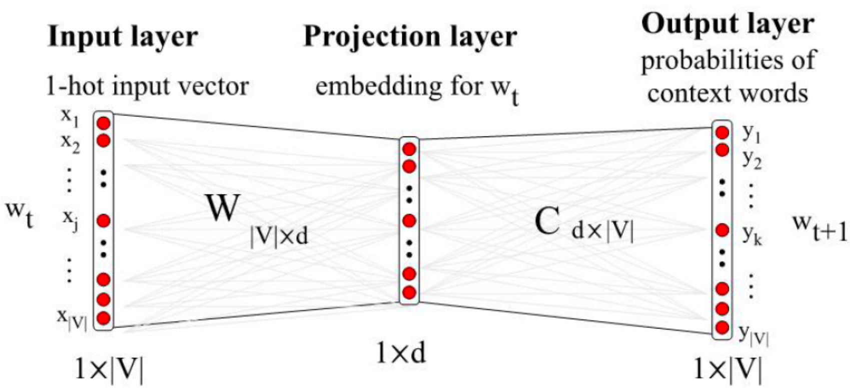
- situation
    - we walk though corpus and are at word corpus(t) at position t
    - index of corpus(t) in the vocabulary is j, so we call it w_j
- goal: predict **corpus**($t+1$) (whose index in vocabulary will be called k)
- actual task: compute $\mathbb{P}(w_k|w_j)$ and return w_k maximizing this probability
- approximate **probability** $\mathbb{P}(w_k|w_j)$ by **similarity** $sim(c_k, v_j)$ (e.g. dot-product)
    - i.e. we multiply one **word vector embedding** with a **context vector embedding**
    - problem: we need a probability, i.e. value in $[0,1]$
    - solution: **softmax normalization**

$$\mathbb{P}(w_k|w_j) = \frac{\exp\left(c_k \cdot v_j\right)}{\sum\limits_{1 \le i \le V} \exp(c_i \cdot v_j)} \tag{10.7}$$

Learning

1) start with some initial vectors (e.g. random)

2) iteratively make the vectors for a word
    - more like the embeddings of its neighbors
    - less like the embeddings of other words



Practical Considerations

- computing the probabilities requires summing over all words in the vocabulary (too expensive)

$$\mathbb{P}(w_k|w_j) = \frac{\exp\left(c_k \cdot v_j\right)}{\sum\limits_{1 \le i \le V} \exp(c_i \cdot v_j)} \tag{10.8}$$

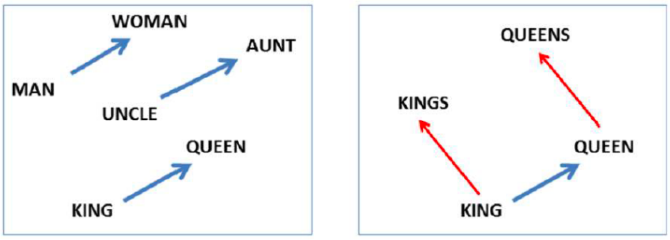
- solution: just sample k negative contexts and sum over these

Properties of Embeddings

- embeddings capture **relational meaning**

  ⇒ vector('king') - vector('man') + vector('woman') ≈ vector('queen')

  ⇒ vector('Paris') - vector('France') + vector('Italy') ≈ vector('Rome')



10.4.2 Brown Clustering

- agglomerative clustering algorithm which clusters words based on near words
  - in the beginning each word has its own cluster
  - in each step pairs of clusters are merged to create larger ones
- word clusters can be turned into kind of vector
- algorithm makes use of **class based** language model in which each word $w \in V$ belongs to a class c with probability $\mathbb{P}(w|c)$
- class based LMs assign a probability to a pair of words as

$$\mathbb{P}(w_i|w_{i-1}) = \mathbb{P}(c_i|c_{i-1}) \cdot \mathbb{P}(w_i|c_i) \tag{10.9}$$

- probability of an entire corpus:

$$\mathbb{P}(corpus|C) = \prod_{i=1}^{n} \mathbb{P}(c_i|c_{i-1}) \cdot \mathbb{P}(w_i|c_i) \tag{10.10}$$

- merge clusters s.t. we minimize the decrease of $\mathbb{P}(corpus|C)$

Brown Clusters as Vectors

- by tracing the order in which clusters are merged, the model builds a binary tree from bottom to top
- each word represented by binary string := path from root to leaf
  - Chairman is 0010, "months" = 01, and verbs = 1