NLP

# Notes from Speech & Language Processing

## Regular Expressions

* an algebraic notation for characterizing a set of strings.
* we have a pattern to search for and a corpus of texts to search through

### Basic RE patterns

// - simplest kind of regular expression is a sequence of simple characters.

* Text

  Description automatically generated

[] – square braces. The string of characters inside the braces specifies a disjunction of characters to match.

* Graphical user interface, text, application

  Description automatically generated
* Text

  Description automatically generated
* Using caret ^ to ignore characters: Text

  Description automatically generated with medium confidence

? - means “the preceding character or nothing”. It is an optional character.

* Graphical user interface, text, application

  Description automatically generated

\* - commonly called the Kleene \* (generally pronounced “cleany star”). “zero or more occurrences of the immediately previous character or regular expression”. /[ab]\*/ will match strings like *aaaa or* *ababab* or *bbbb*.

+ *Kleene +,* which means “one or more occurrences of the immediately preceding character or regular expression”. /[0-9]+/ is the normal way to specify “a sequence of digits”.

. wildcard expression, that matches any single character (except a carriage return)

* Text

  Description automatically generated
* suppose we want to find any line in which a particular word, for example, aardvark, appears twice. We can specify this with the regular expression /aardvark.\*aardvark/

Anchors - the caret ˆ and the dollar sign $

* The caret ˆ has three uses:
* to match the start of a line,
* to indicate a negation inside of square brackets,
* and just to mean a caret.
* The dollar sign $ matches the end of a line.
* the pattern $ is a useful pattern for matching a space at the end of a line, and /ˆThe dog\.$/ matches a line that contains only the phrase The dog. (We have to use the backslash here since we want the . to mean “period” and not the wildcard.

Table

Description automatically generated

* \b matches a word boundary - /\bthe\b/ matches the word the but not the word other
* \B matches a non-boundary.

### Disjunction, Grouping, and Precedence­­

* the **disjunction operator**, also called the pipe symbol |. The pattern /cat|dog/ matches either the string cat or the string dog.
* How can I specify both guppy and guppies?
* /guppy|ies/, cant work because that would match only the strings guppy and ies.
* Precedence - To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators ( and )
* /gupp(y|ies)/ would specify that we meant the disjunction only to apply to the suffixes y and ies.
* Matching repeated text in a form, example - *Column 1 Column 2 Column 3*
*  will not match any column, instead it will match a single column followed by any number of spaces!
* The star here applies only to the space  that precedes it, not to the whole sequence.
* /(Column [0-9]+ \*)\*/ to match the word Column, followed by a number and optional spaces, the whole pattern repeated zero or more times.
* Operator precedence

Graphical user interface

Description automatically generated with low confidence

* Reducing the overall error rate for an application thus involves two antagonistic efforts:

• Increasing precision (minimizing false positives)

• Increasing recall (minimizing false negatives)

### More Operators

.{} – gives the counted number of occurrences

* /{n,m}/ specifies from n to m occurrences of the previous char or expression, and /{n,}/ means at least n occurrences of the previous expression.

Text

Description automatically generated with medium confidence

Text

Description automatically generated

Text

Description automatically generated

### More complex RE

We plan to denote the dollars here:

* /$[0-9]+/ 🡪 $ character has a different function here than the end-of-line f() we discussed earlier.
*  🡪 deals with the decimal values now. Matches $199.99 but not $199
*  🡪 but this allows very high prices like $19999999.99, so to set some limits, we use the next step.
*  🡪 does the work.

### Substitutionn, Capture groups & ELIZA

**Substitution** 🡪



* Here the \1 will be replaced by whatever string matched the first item in parentheses. So, this will match *the* ***bigger*** *they were, the* ***bigger*** *they will be*

but not *the* ***bigger*** *they were, the* ***faster*** *they will be*.

**Capture Group** 🡪



will match *the* ***faster*** *they* ***ran****, the* ***faster*** *we* ***ran***,

but not *the* ***faster*** *they* ***ran****, the* ***faster*** *we* ***ate***.

* This use of parentheses to store a pattern in memory is called a capture group.
* Every time a capture group is used (i.e., parentheses surround a pattern), the resulting match is stored in a numbered register.

Parentheses thus have a double function in regular expressions; they are used to group terms for specifying the order in which operators should apply, and they are used to capture something in a register.

Occasionally we might want to use parentheses for grouping, but don’t want to capture the resulting pattern in a register. In that case we use a **non-capturing group**, which is specified by putting the commands ?: after the open paren, in the form (?: pattern ).

**Non-Capturing Groups** 🡪



will match *some cats like some cats*

but not *some cats like some a few*.

Using these in ELISA:

Text

Description automatically generated

### Lookahead & Assertions

These lookahead assertions make use of the (? syntax that we saw in the previous section for non-capture groups. The operator (?= *pattern*) is true if pattern occurs, but is zero-width, i.e. the match pointer doesn’t advance. The operator (?! *pattern*) only returns true if a pattern does not match, but again is zero-width and doesn’t advance the cursor.

* For example suppose we want to match, at the beginning of a line, any single word that doesn’t start with “Volcano”. We can use negative lookahead to do this:

/ˆ(?!Volcano)[A-Za-z]+/

## Words

“I do uh main- mainly business data processing”

Disfluencies 🡪

Fragment – The broken-off words (main-)

Filler/ filler pauses – words like uh and um (uh)

* in speech recognition in predicting the upcoming word, because they may signal that the speaker is restarting the clause or idea, and so for speech recognition they are treated as regular words.
* Depends on context whether to use them or ignore.

Lemma 🡪

A lemma is a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense

Wordform 🡪

The wordform is the full inflected or derived form of the word.

How many words are there? We can find using two ways:

Word types 🡪

Types are the number of distinct words in a corpus; if the set of words in the vocabulary is V, the number of types is the vocabulary size |V|.

Word Tokens 🡪

Tokens are the total number N of running words.

Herden’s Law/ Heaps’ law🡪

The larger corpora we have, more word types we find. So there is relation between |V| & N.

 where k and β are positive constants, and 0 < β < 1

β – depends on the corpus size and genre.

## Corpora

The world has 7097 languages. It is important to test algorithms on more than one language, and particularly on languages with different properties.

Even when algorithms are developed beyond English, they tend to be developed for the official languages of large, industrialized nations (Chinese, Spanish, Japanese, German etc.).

The same language can have multiple varities based on the *region* and *social groups*.

**Code switching** - to use multiple languages in a single communicative act.

Text also reflects the demographic characteristics of the writer (or speaker): their age, gender, race, socioeconomic class can all influence the linguistic properties of the text we are processing.

Language changes over time, and for some languages we have good corpora of texts from different historical periods.

Building corpus 🡪 building a datasheet.

Datasheet:

Motivation.

Situation.

Language variety.

Speaker demographics

Collection process

Annotation process

Distribution.

## Text Normalization

At least three tasks are commonly applied as part of any normalization process:

1. Tokenizing (segmenting) words

2. Normalizing word formats

3. Segmenting sentences

#### Word Tokenization - the task of segmenting running text into words.

We often want to break off punctuation as a separate token; commas are a useful piece of information for parsers, periods help indicate sentence boundaries. But we’ll often want to keep the punctuation that occurs word internally, in examples like m.p.h., Ph.D., AT&T, and cap’n. Special characters and numbers will need to be kept in prices ($45.55) and dates (01/02/06); we don’t want to segment that price into separate tokens of “45” and “55”. And there are URLs (http://www.stanford.edu), Twitter hashtags (#nlproc), or email addresses (someone@cs.colorado.edu). Number expressions introduce other complications as well; while commas normally appear at word boundaries, commas are used inside numbers in English, every three digits: 555,500.50. Languages, and hence tokenization requirements, differ on this; many continental European languages like Spanish, French, and German, by contrast, use a comma to mark the decimal point, and spaces (or sometimes periods) where English puts commas, for example, 555 500,50.

A tokenizer can also be used to expand **clitic** contractions that are marked by apostrophes, for example, converting *what’re* to the two tokens *what are*, and *we’re* to *we are*.

##### Penn Treebank tokenization standard

used for the parsed corpora (treebanks) released by the Linguistic Data Consortium (LDC), the source of many useful datasets.

This standard separates out clitics (doesn’t becomes does plus n’t), keeps hyphenated words together, and separates out all punctuation.

Graphical user interface, text

Description automatically generated

* In practice, tokenization needs to be run before any other language processing, it needs to be very fast.
* The standard method for tokenization is therefore to use deterministic algorithms based on RE compiled into very efficient finite state automata.

Text

Description automatically generated

#### Byte-Pair Encoding for Tokenization

Instead of defining tokens as words we can use our data to automatically tell us what the tokens should be.

* NLP algorithms often learn some facts about language from one corpus (a training corpus) and then use these facts to make decisions about a separate test corpus and its language. Thus if our training corpus contains, say the words *low, new, newer*, but not *lower*, then if the word *lower* appears in our test corpus, our system will not know what to do with it.
* To deal with this unknown word problem, modern tokenizers often automatically induce sets of tokens that include tokens smaller than words, called **subwords**.
* Subwords can be arbitrary substrings, or they can be meaning-bearing units like the morphemes -est or -er. (A morpheme is the smallest meaning-bearing unit of a language; for example the word *unlikeliest* has the morphemes *un-*, likely, and *-est*.)
* Two parts of tokenizer: a **token learner**, and a **token segmenter**.
* a **token learner** - takes a raw training corpus and induces a vocabulary, a set of tokens.
* a **token segmenter** - takes a raw test sentence and segments it into the tokens in the vocabulary.
* 3 algorithms are used:
* **Byte-pair encoding:** BPE token learner begins with a vocabulary that is just the set of all individual characters. It then examines the training corpus, chooses the two symbols that are most frequently adjacent (say ‘A’, ‘B’), adds a new merged symbol ‘AB’ to the vocabulary, and replaces every adjacent ’A’ ’B’ in the corpus with the new ‘AB’. It continues to count and merge, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus a parameter of the algorithm

The resulting vocabulary consists of the original set of characters plus k new symbols.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**Text

Description automatically generated with medium confidence**

* **unigram language modelling.**
* **WordPiece.**
* **SentencePiece** library that includes implementations of the first two of the three.

#### Word Normalization, Lemmatization and Stemming

**Word normalization** 🡪

Putting words/tokens in a standard format, choosing a single normal form for words with multiple forms. [*USA* and *US* or *uh-huh* and *uhhuh*.]

**Case folding** 🡪

Mapping everything to lowercase means that Woodchuck and woodchuck are represented identically, which is very helpful for generalization in many tasks

Not done in case of sentiment analysis and other text classification tasks, information extraction, and machine translation, in fact, case can be quite helpful and case folding is generally not done.

**Lemmatization** 🡪

task of determining that two words have the same root, despite their surface differences. The words *am*, *are*, and *is* have the shared lemma *be*. the words *dinner* and *dinners* both have the lemma *dinner.*

* How to do lemmatization?
  + morphological parsing - is the study of the way words are built up from smaller meaning-bearing units called **morphemes.**
  + Morphemes has two classes:
    - Stems: contains the main meaning.
    - Affixes: adds additional meaning of various kinds.
  + Example: *cats* has 2 morphemes – *cat* and *-s.*

##### The Porter Stemmer

* simpler but cruder method, which mainly consists of chopping off word-final affixes.
* This naive version of morphological analysis is called stemming.

Text

Description automatically generated

* Following rewrite rules are used for modification

Text

Description automatically generated

##### Sentence Segmentation

* The most useful cues for segmenting a text into sentences are punctuation, like periods, question marks, and exclamation points.
* Question marks and exclamation points are relatively unambiguous markers of sentence boundaries.
* Periods, on the other hand, are more ambiguous. “.” is ambiguous between a sentence boundary marker and a marker of abbreviations like *Mr*. or *Inc.*
* So, sentence tokenization and word tokenization may be addressed jointly.

## Minimum Edit Distance

* measuring how similar two strings are.
* graffe & giraffe – only one letter difference.



* **Edit distance** gives us a way to quantify both of these intuitions about string similarity. More formally, the **minimum edit distance** between two strings is definedas the minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another.
* Given two sequences, an **alignment** is a correspondence between substrings of the two sequences.

Text

Description automatically generated with medium confidence

* Levenshtein distance –

Assigning weights to these operations. Each operation has cost 1. So above the cost is 5.

Another alternate to this:

* each insertion or deletion has a cost of 1
* substitutions are not allowed.

This above is equivalent to allowing substitution, but giving each substitution a cost of 2 since any substitution can be represented by one insertion and one deletion

Diagram, schematic

Description automatically generated

* The Minimum Edit Distance Algorithm –

A picture containing diagram

Description automatically generated

The space of all possible edits is enormous, so we can’t search naively.

However, lots of distinct edit paths will end up in the same state (string), so rather than recomputing all those paths, we could just remember the shortest path to a state each time dynamic we saw it. We can do this by using dynamic programming.

**DP** -- that apply a table-driven method to solve problems by combining solutions to sub-problems. Algos using DP - *Viterbi algorithm*. *CKY algo*.

Text

Description automatically generated

Text

Description automatically generated

* Definition of MEDA –

Given two strings, the source string X of length n, and target string Y of length m, we’ll define D[i, j] as the edit distance between X[1..i] and Y[1.. j], i.e., the first i characters of X and the first j characters ofY. The edit distance between X and Y is thus D[n,m]

A picture containing text, antenna

Description automatically generated

* MEDA is useful for finding error corrections.
* Helps in finding the alignment between two string.

* Alignment –

In speech recognition, minimum edit distance alignment is used to compute the word error rate.

Alignment plays a role in machine translation, in which sentences in a parallel corpus (a corpus with a text in two languages) need to be matched to each other.

Text

Description automatically generated

Table

Description automatically generated with medium confidence

# N – Gram Language Models

Probabilities are also important for augmentative and alternative communication systems.

People often use such AAC devices if they are physically unable to speak or sign but can instead use eye gaze or other specific movements to select words from a menu to be spoken by the system. Word prediction can be used to suggest likely words for the menu.

Models that assign probabilities to sequences of words are called language models or LMs.

## N-gram

* sequence of n words.
* the task of computing P(w|h), the probability of a word w given some history h.
* Suppose the history h is “*its water is so transparent that*” and we want to know the probability that the next word is *the*:



* + Method 1 — estimate this probability is from relative frequency counts.

count the number of times we see its water is so transparent that, and count the number of times this is followed by the. This would be answering the question “Out of the times we saw the history h, how many times was it followed by the word w”, as follows:

Text, table

Description automatically generated with medium confidence

* + - Drawbacks –
      * Language is creative.
      * New sentences are created all time.
      * we won’t always be able to count entire sentences
  + Method 2 –

To represent the probability of a particular random variable *Xi* taking on the value “the”, or P(Xi = “the”), we will use the simplification P(the).

We’ll represent a sequence of N words either as w1 . . .wn or w1:n (so the expression w1:n−1 means the string w1,w2, ...,wn−1).

For the joint probability of each word in a sequence having a particular value P(X =w1,Y = w2,Z = w3, ...,W = wn) we’ll use P(w1,w2, ...,wn).

To compute probabilities of entire sequences like P(w1,w2, ...,wn), we can decompose this probability using the **chain rule of probability**:

Text

Description automatically generated with low confidence

The bigram model, for example, approximates the probability of a word given all the previous words P(wn|w1:n−1) by using only the conditional probability of the preceding word P(wn|wn−1).



* + - The assumption that the probability of a word depends only on the previous word is called a **Markov assumption.**
    - Markov model are probabilistic models that assume we can predict the probability of future unit without looking too far in the past. E.g. bigram & trigram.
    - General equation of n-gram:



* **Maximum likelihood estimation** or **MLE** -- estimate these bigrams or n-gram probabilities.
  + We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1.
  + E.g. to compute a particular bigram probability of a word y given a previous word x, we’ll compute the count of the bigram C(xy) and normalize by the sum of all the bigrams that share the same first word x:

Text

Description automatically generated

* + Instead of focusing on the probability of word in the whole corpus, we use something as shown:

Table

Description automatically generated

This can provide probability as follows:

Table

Description automatically generated

When using higher n-gram models, we have to consider more probabilities, and multiplication of these can cause numeric underflow.

So, we use logarithms because we get numbers that are not as small. Adding in log space is equivalent to multiplying in linear space, so we combine log probabilities by adding them. The result of doing all computation and storage in log space is that we only need to convert back into probabilities if we need to report them at the end; then we can just take the exp of the logprob:



## Evaluating the language models

**Extrinsic evaluation:**

to evaluate the performance of a language model - embed it in an application and measure how much the application improves. This is an end-to-end evaluation.

Unfortunately, running big NLP systems end-to-end is often very expensive.

**Intrinsic evaluation:**

An intrinsic evaluation metric is one that measures the quality of a model independent of any application.

So if we are given a corpus of text and want to compare two different n-gram models,

* we divide the data into training and test sets,
* train the parameters of both models on the training set,
* and then compare how well the two trained models fit the test set.

“fit the test set” 🡪 whichever model assigns a higher probability to the test set—meaning it more accurately predicts the test set—is a better model.

### Perplexity (PP)

In practice we don’t use raw probability as our metric for evaluating language models, but a variant called perplexity. For a test set W = w1w2 . . .wN,

Diagram

Description automatically generated using chain rule, we get:

Diagram

Description automatically generated computing PP(W) with a bigram language model we get:

Diagram, schematic

Description automatically generated

Since there is the inverse, the higher the conditional probability of the word sequence, the lower the PP.

PP can also be represented as, weighted average branching factor of a language. The branching factor of a language is the number of possible next words that can follow any word. E.g. for the task of recognising the digits 0-9 given each digit occurs with = probability (1/10). The PP for this mini-language is 10 as shown below:

Diagram

Description automatically generated

But suppose when 0 is more frequent than others, the perplexity of this test set is lower since most of the time the next num will be zero which is very predictable.

Thus, although the branching factor is still 10, the perplexity or weighted branching factor is smaller

PP is also closely related to the information-theoretic notion of entropy.

Text

Description automatically generated with medium confidenceThe table below shows the perplexity of a 1.5 million word WSJ test set according to each of these grammars. the more information the n-gram gives us about the word sequence, the lower the perplexity.

The perplexity of two language models is only comparable if they use identical vocabularies.

An (intrinsic) improvement in perplexity does not guarantee an (extrinsic) improvement in the performance of a language processing task like speech recognition or machine translation. Nonetheless, because perplexity often correlates with such improvements, it is commonly used as a quick check on an algorithm. But a model’s improvement in perplexity should always be confirmed by an end-to-end evaluation of a real task before concluding the evaluation of the model.

## Sampling sentences from a language model

Sampling from a distribution means to choose random points according to their likelihood.

A picture containing timeline

Description automatically generated

## Generalization and Zeros

Text

Description automatically generated

Text

Description automatically generated

Since, both of them are trained on English-like sentences but different corpora. There is no overlap in the generated sentences. So statistical models are likely to be pretty useless as predictors if the training set and test sets are different as Shakespeare and WSJ.

So choice of genre is important, we pick the training corpus based upon use-case.

Matching genres and dialects is still not sufficient. Our models may still be subject to the problem of **sparsity**.

Suppose our training corpus had these words:

A black and white sign

Description automatically generated with low confidenceBut suppose our test set has phrases like:

1. denied the offer
2. denied the loan

Our model will incorrectly estimate that the P(offer|denied the) is 0! This is *zero probability n-grams* problem. This can occur due to two reasons:

* their presence means we are underestimating the probability of all sorts of words that might occur
* if the probability of any word in the test set is 0, the entire probability of the test set is 0

In such cases we cannot calculate the PP, as we can’t divide by 0!

### Unknown words

In a **closed vocabulary** system the test set can only contain words from this lexicon, and there will be no unknown words.

In other cases we have to deal with unseen words, (unknown words) ~ out of vocabulary (OOV) words.

The percentage of OOV words that appear in the test set is called the OOV rate.

An open vocabulary system is one in which we model these potential unknown words in the test set by adding a pseudo-word called <UNK>

**Ways to train the probabilities of the unknown word model <UNK>**

* turn the problem back into a closed vocabulary one by choosing a fixed vocabulary in advance:

Text

Description automatically generated

* in situations where we don’t have a prior vocabulary in advance, is to create such a vocabulary implicitly, replacing words in the training data by <UNK> based on their frequency.

## Smoothing

What do we do with words that are in our vocabulary (they are not unknown words) but appear in a test set in an unseen context.

To keep a language model from assigning zero probability to these unseen events, we’ll have to shave off a bit of probability mass from some more frequent events and give it to the events we’ve never seen. This modification is called **smoothing** or **discounting**.

### Laplace Smoothing (LS) – add one smoothing

* add one to all the n-gram counts, before we normalize them into probabilities. All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on.
* LS performance is not as good enough to be used in the modern n-gram models.
* wi - word

ci - count

N - total number of word tokens

V – words in vocab

unsmoothed maximum likelihood estimates of the unigram probability

Text

Description automatically generated with low confidence

* Since, each word in the vocab was incremented, ,we need to adjust the denominator to take into account extra V obs.

Logo

Description automatically generated

* Instead of changing numerator or denominator, it is convenient to describe how a smoothing algorithm affects the numerator, by defining an adjusted count c∗.

Diagram

Description automatically generated

We can now turn ci\* into a probability Pi∗ by normalizing by N.

* A related way to view smoothing is as discounting (lowering) some non-zero counts in order to get the probability mass that will be assigned to the zero counts.
* Thus, instead of referring to the discounted counts c∗, we might describe a smoothing algorithm in terms of a relative discount dc, the ratio of the discounted counts to the original counts:

A picture containing text, watch

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

* It is often convenient to reconstruct the count matrix so we can see how much a smoothing algorithm has changed the original counts, these adjusted counts can be computed by -

Diagram

Description automatically generated

Table

Description automatically generated

The sharp change in counts and probabilities occurs because too much probability mass is moved to all the zeros.

### Add k-smoothing

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events.

Instead of adding 1 to each count, we add a fractional count k (.5? .05? .01?).

Text

Description automatically generated with medium confidence

Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a **devset**.

Applications: text classification

Bad with: language modeling, generating counts with poor variances and often inappropriate discounts.

### Backoff and Interpolation

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on.

Sometimes using less context is a good thing, helping to generalize more for contexts that the model hasn’t learned much about.

There are two ways to use this n-gram “hierarchy”:

* In **backoff**, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only “back off” to a lower-order n-gram if we have zero evidence for a higher-order n-gram.
* By contrast, in **interpolation**, we always mix the probability estimates from all the n-gram estimators, weighing and combining the trigram, bigram, and unigram counts.

**In simple linear interpolation,** we combine different order n-grams by linearly interpolating them.

Text

Description automatically generated

here, λs must sum up to 1.

In a slightly more sophisticated version of linear interpolation, each λ weight is computed by conditioning on the context.

If we have particularly accurate counts for a particular bigram, we assume that the counts of the trigrams based on this bigram will be more trustworthy, so we can make the λs for those trigrams higher and thus give that trigram more weight in the interpolation.

Text, letter

Description automatically generated

λs can be set 🡪 Both the simple interpolation and conditional interpolation λs are learned from a **held-out corpus**.

A held-out corpus is an additional training corpus that we use to set hyperparameters like these λ values, by choosing the λ values that maximize the likelihood of the held-out corpus.

* So we fix n-gram probabilities and then search for the λ values that—when plugged into Eq. 3.26—give us the highest probability of the held-out set.
* Other ways to calculate the optimal set of λs;
  + EM algorithm

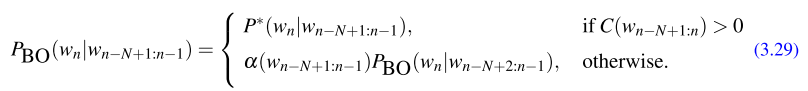
an iterative learning algorithm that converges on locally optimal λs

**In a backoff n-gram model**, if the n-gram we need has zero counts, we approximate it by backing off to the (N-1)-gram. We continue backing off until we reach a history that has some counts.

In order for a backoff model to give a correct probability distribution, we have to discount the higher-order n-grams to save some probability mass for the lower order n-grams.

In addition to this explicit discount factor, we’ll need a function α to distribute this probability mass to the lower order n-grams.

This kind of backoff with discounting is also called **Katz backoff**.

 Katz backoff is often combined with a smoothing method called **Good-Turing**.

The combined Good-Turing backoff algorithm involves quite detailed computation for estimating the Good-Turing smoothing and the P∗ and *α* values.

## Kneser-Ney Smoothing

One of the best performing n-gram smoothing methods is the interpolated Kneser-Ney algorithm.

Uses concept of– **absolute discounting**.

Consider an n-gram that has count 4. We need to discount this count by some amount. But how much should we discount it? Church and Gale’s clever idea was to look at a held-out corpus and just see what the count is for all those bigrams that had count 4 in the training set.

Table

Description automatically generated

Note that except for the held-out counts for 0 and 1, all other bigram counts could be estimated pretty well by subtracting 0.75 from the count in the training set!

Absolute discounting formalizes this intuition by subtracting the fixed absolute discount d from each count. since we have good estimates already for the very high counts, a small discount d won’t affect them much. It will mainly modify the smaller counts, for which we don’t necessarily trust the estimate anyway.

Text, letter

Description automatically generated with medium confidence

The first term is the discounted bigram, and the second term is the unigram with an interpolation weight λ.

* instead of P(w), which answers the question “How likely is w?”, we’d like to create a unigram model that we might call PCONTINUATION, which answers the question “How likely is w to appear as a novel continuation?”
* Every bigram type was a novel continuation the first time it was seen. The number of times a word w appears as a novel continuation can be expressed as:



Turning this count to probability now –



## Huge Language Models and Stupid Backoff

**COCA** is a balanced corpora, meaning that it has roughly equal numbers of words from different genres: web, newspapers, spoken conversation transcripts, fiction, and so on, drawn from the period 1990-2019.

**Efficiency**

* instead of storing each word as string – 64-bit hash number, with the words themselves stored on disk.
* Probabilities are generally quantized using only 4-8 bits (instead of 8-byte floats)
* n-grams are stored in reverse tries
* Pruning can be used to shrink the n-gram LM, based on counts greater than threshold, or using entropy to prune the less important n-grams; build approximate language models using techniques like **Bloom filters.**
* efficient language model toolkits like **KenLM** use sorted arrays, efficiently combine probabilities and backoffs in a single value, and use merge sorts to efficiently build the probability tables in a minimal number of passes through a large corpus.
* Above toolkits show that with very large language models a much simpler algorithm may be sufficient. The algorithm is called **stupid backoff**. Stupid backoff gives up the idea of trying to make the language model a true probability distribution. There is no discounting of the higher-order probabilities. If a higher-order n-gram has a zero count, we simply backoff to a lower order n-gram, weighed by a fixed (context-independent) weight. This algorithm does not produce a probability distribution as S:

Text, letter

Description automatically generated

## Advanced: Perplexity’s Relation to Entropy

Entropy is a measure of information. Given a random variable X ranging over whatever we are predicting (words, letters, parts of speech, the set of which we’ll call χ) and with a particular probability function, call it p(x), the entropy of the random variable X is

Text

Description automatically generated with medium confidence

To save info about 8 numbers, we can use 3 bits of space. A better way is using the entropy.

Suppose that the spread is the actual distribution of the bets placed and that we represent it as the prior probability of each horse as follows:

Text, table

Description automatically generated with medium confidenceThe entropy of the random variable X that ranges over horses gives us a lower bound on the number of bits and is:

Text, letter

Description automatically generated

A code that averages 2 bits per race can be built with short encodings for more probable horses, and longer encodings for less probable horses.

For example, we could encode the most likely horse with the code 0, and the remaining horses as 10, then 110, 1110, 111100, 111101, 111110, and 111111.

**Concept of entropy in sequences.**

computing the entropy of some sequence ofwordsW= {w1,w2, . . . ,wn}.

We could define the entropy rate (we could also think of this as the per-wordentropy) as the entropy of this sequence divided by the number of words:

Text

Description automatically generated

But to measure the true entropy of a language, we need to consider sequences of infinite length. If we think of a language as a stochastic process L that produces a sequence of words, and allow W to represent the sequence of words w1, . . . ,wn, then L’s entropy rate H(L) is defined as:

Text

Description automatically generated

if the language is regular in certain ways (to be exact, if it is both stationary and ergodic),

A picture containing text, watch, gauge

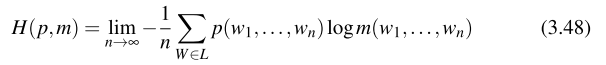
Description automatically generated

That is, we can take a single sequence that is long enough instead of summing over all possible sequences. The intuition of the Shannon-McMillan-Breiman theorem is that a long-enough sequence of words will contain in it many other shorter sequences and that each of these shorter sequences will reoccur in the longer sequence according to their probabilities.

A stochastic process is said to be **stationary** if the probabilities it assigns to a sequence are invariant with respect to shifts in the time index. In other words, the probability distribution for words at time t is the same as the probability distribution at time t +1.

Markov models, and hence n-grams, are stationary. For example, in a bigram, Pi is dependent only on Pi−1. So if we shift our time index by x, Pi+x is still dependent on Pi+x−1.

The **cross-entropy** is useful when we don’t know the actual probability distribution p that generated some data. It allows us to use some m, which is a model of p (i.e., an approximation to p). The cross-entropy of m on p is defined by-





for entropy, we can estimate the cross-entropy of a model m on some distribution p by taking a single sequence that is long enough instead of summing over all possible sequences.

What makes the cross-entropy useful is that the cross-entropy H(p,m) is an upper bound on the entropy H(p).



The more accurate the model m will be, the cross entropy H(p,m) will be closer to the true entropy H(p). Difference between these denotes the accuracy of model.

Lower cross entropy 🡪 more accurate the model.

# Naïve Bayes and Sentiment Classification

Here we will apply naïve Bayes algo to **text categorization**, the task of assigning a label or sentiment analysis category to an entire text or document.

We focus on one common text categorization task, **sentiment analysis**, the extraction of sentiment.

**Spam detection** is another important commercial application, the binary classification task of assigning an email to one of the two classes spam or not-spam.

In the **supervised** situation we have a training set of N documents that have each been hand-labeled with a class: (d1,c1), ...., (dN,cN). Our goal is to learn a classifier that is capable of mapping from a new document d to its correct class c ∈C. A **probabilistic classifier** additionally will tell us the probability of the observation being in the class.

**Generative classifiers** like naive Bayes build a model of how a class could generate some input data.

**Discriminative classifiers** like logistic regression instead learn what features from the input are most useful to discriminate between the different possible classes.

While discriminative systems are often more accurate and hence more commonly used,

## Naïve Bayes Classifiers

**multinomial naive Bayes classifier**, so called because it is a Bayesian classifier that makes a simplifying (naive) assumption about how the features interact.

**bag-of-words** is an unordered set of words with their position ignored, keeping only their frequency in the document.

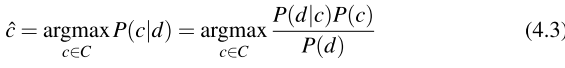
Diagram, text

Description automatically generated

Naive Bayes is a probabilistic classifier, meaning that for a document d, out of all classes c ∈C the classifier returns the class ˆc which has the maximum posterior probability given the document. In Eq. 4.1 we use the hat notation ˆ to mean “our estimate of the correct class”.



Substituting the Bayes’ rule we get:



In this eq 4.3, since the doc remains same, the probability P(d) remains same, so we can maximise a simpler formula:  
 

For classification, we compute ^c given doc d by choosing the class with has the highest product of two probabilities:

* the **prior** probability of the class P(c)
* the **likelihood** of the document P(d|c) 🡪 representing doc d as set of features f1, f2, f3..fn.

Text

Description automatically generated 🡪 Text

Description automatically generated

Simplifying the latter eq. as calculating possible combination of features is intensive task, we use two ways:

1. bag of words assumption: position doesn’t matter.
2. **naive Bayes assumption**: conditional independence assumption that the probabilities P( fi|c) are independent given the class c.

Text

Description automatically generated

To apply naïve Bayes classifier to text, we consider the word positions, by indexing the word position in document.

Text

Description automatically generated with low confidence



Eq. 4.10 computes the predicted class as a linear function of input features. Classifiers that use a linear combination of the inputs to make a classification decision —like naive Bayes and also logistic regression— are called **linear classifiers**.

## Training the Naive Bayes Classifier

How can we learn the probabilities P(c) and P( fi|c)?

For the class prior P(c) we ask what % of the documents in our training set are in each class c.

Nc - number of documents in our training data with class c

Ndoc - total number of documents. Then:

Background pattern

Description automatically generated with low confidence

P(wi|c), which we compute as the fraction of times the word wi appears among all words in all documents of topic c. We first concatenate all documents with category c into one big “category c” text. Then we use the frequency of wi in this concatenated document to give a maximum likelihood estimate of the probability:

Text

Description automatically generated