Regular expressions:

- for pre-processing or text formatting
- Necessary for data analysis
- Letters inside square brackets []
- Ranges using the dash [A-Z]

  Carat as first character in [] negates the list
  - Note: Carat means negation only when it's first in []
  - Special characters (., *, +, ?) lose their special meaning inside []

Pattern	Matches	Examples
[^A-Z]	Not an upper case letter	Oyfn pripetchik
[^Ss]	Neither 'S' nor 's'	I have no exquisite reason"
[^.]	Not a period	Our resident Djinn
[e^]	Either e or ^	Look up ^ now

-

Pattern	Expansion	Matches	Examples
\d	[0-9]	Any digit	Fahreneit 451
\D	[^0-9]	Any non-digit	Blue Moon
\w	[a-zA-Z0-9_]	Any alphanumeric or _	Daiyu
\W	[^\w]	Not alphanumeric or _	Look!
\s	[\r\t\n\f]	Whitespace (space, tab)	Look up
\S	[^\s]	Not whitespace	Look up

- the backslash, to make even more aliases. So \d means "any one digit", \s means whitespace and \w means any alphanumeric (plus the underbar, slightly confusing). Using the capital letter means negated, so \D means any non-digit.
- the square brackets choose between individual characters, but the pipe chooses between strings of characters. So this disjunction means "either the string groundhog or the string woodchuck". If we did want to speciy disjunctions of

single letters we could use either the square brackets or the pipe

Pattern	Matches	Examples
beg.n	Any char	begin begun beg3n beg n
woodchucks?	Optional s	woodchuck woodchucks
to*	0 or more of previous char	t to too tooo
to+	1 or more of previous char	to too tooo toooo

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

  The carat matches the start of the line, and the $ matches the end of the line. So carat [A-Z] means "a capital letter at the start of a line". Dot means any character. Backlash period means a literally period.

- Reducing error in NLP:

  Reducing the error rate for an application often involves two antagonistic efforts:
  - Increasing coverage (or *recall*) (minimizing false negatives).
  - Increasing accuracy (or *precision*) (minimizing false positives)

- the python "S" command can be used to change a string matched by a regex to the substitute, another string.

- Say we want to put angles around all numbers:
  the 35 boxes → *the <35> boxes*
- Use parens () to "capture" a pattern into a numbered register (1, 2, 3…)
- Use \1 to refer to the contents of the register

```
s/([0-9]+)/<\1>/
```

- Parentheses have a double function: grouping terms, and capturing

  Non-capturing groups: add a ?: after paren:

  ```
  /(?:some|a few) (people|cats) like some \1/
  ```

  matches
  - some cats like some cats

  but not
  - some cats like some some

- (?= pattern) is true if pattern matches, but is **zero-width; doesn't advance character pointer**
- (?! pattern) true if a pattern does not match

The operator (?= pattern) is true if the pattern occurs, but is zero-width, meaning the match pointer doesn't advance. And the negative lookahead, ?! pattern only returns true if a pattern does not match, but again is zero-width. Negative lookahead is commonly used when we are parsing some complex pattern but want to rule out a special case.

Words and Corpora:

"I do uh main- mainly business data processing"
- Fragments, filled pauses

"Seuss's cat in the hat is different from other cats!"
- **Lemma**: same stem, part of speech, rough word sense
  - cat and cats = same lemma
- **Wordform**: the full inflected surface form
  - cat and cats = different wordforms

- We call things like "main" here a fragment, and we call "uh" and "um" filled pauses. So for certain applications, like speech applications, we might want to be counting these.
- We could count word types, the number of unique words that occur in the sentence. By that count we only count "the" once, even though it appears twice. Word tokens we're counting every word token on the page, so the two "the"s count twice.

N = number of tokens

V = vocabulary = set of types, **|*V*|** is size of vocabulary

Heaps Law = Herdan's Law = $|V| = kN^{\beta}$ where often .67 < β < .75

i.e., vocabulary size grows with > square root of the number of word tokens

- Corpora vary along dimension like: Language, Variety, Code switching, Genre, Author Demographics

Word tokenization:

- Tokenizing (segmenting) words, Normalizing word formats, Segmenting sentences
- space-based tokenization: "tr" command. (Given a text file, output the word tokens and their frequencies)

- in Chinese it's common to just treat each character (zi) as a token. (single-character segmentation)
- Subword tokenization => Most subword algorithms are run inside space-separated tokens. So we commonly first add a special end-of-word symbol '__' before space in training corpus:
  1. Byte-Pair Encoding (BPE):

     function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

     $V \leftarrow$ all unique characters in C # initial set of tokens is characters
     for $i = 1$ **to** k **do** # merge tokens til k times
     $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
     $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
     $V \leftarrow V + t_{NEW}$ # update the vocabulary
     Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
     return V

     On the test data, run each merge learned from the training data.
     This method usually includes frequent words/subwords(Which are often morphemes, A **morpheme** is the smallest meaning-bearing unit of a language).
  2. Unigram language modeling tokenization:
  3. WordPiece

- lemmatization:

  Lemmatization is done by Morphological Parsing

  Morphemes:
  - The small meaningful units that make up words
  - **Stems**: The core meaning-bearing units
  - **Affixes**: Parts that adhere to stems, often with grammatical functions

  Representing all words as their lemma, their shared root.

Porter Stemmer

Based on a series of rewrite rules run in series
- A cascade, in which output of each pass fed to next pass

Some sample rules:

$$\begin{aligned} \text{ATIONAL} &\rightarrow \text{ATE} \quad \text{(e.g., relational} \rightarrow \text{relate)} \\ \text{ING} &\rightarrow \epsilon \quad \text{if stem contains vowel (e.g., motoring} \rightarrow \text{motor)} \\ \text{SSES} &\rightarrow \text{SS} \quad \text{(e.g., grasses} \rightarrow \text{grass)} \end{aligned}$$

Minimum Edit Distance:

- Using in Spell correction, Computational Biology, Machine Translation, Information Extraction, Speech Recognition
  - The minimum edit distance between two strings
  - Is the minimum number of editing operations
    - Insertion
    - Deletion
    - Substitution
- **Initial state**: the word we're transforming
- **Operators**: insert, delete, substitute
- **Goal state**: the word we're trying to get to
- **Path cost**: what we want to minimize: the number of edits

Defining Min Edit Distance (Levenshtein)

Initialization

$D(i,0) = i$

$D(0,j) = j$

Recurrence Relation:

For each $i = 1 \ldots M$

  For each $j = 1 \ldots N$

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

Termination:

$D(N,M)$ is distance

N-grams:

- Using in Spell correction, Machine Translation, Speech Recognition, Summarization, question-answering.
  - A model that computes either of these:

    P(W) or $P(w_n|w_1,w_2...w_{n-1})$ is called a **language model**.
  - Better: **the grammar** But **language model** or **LM** is standard
-
- Computing the probabilistic of a sentence, using chain rule: $P(w_1 w_2 \ldots w_n) = \prod_i P(w_i \mid w_1 w_2 \ldots w_{i-1})$ but we cannot count as Too many possible sentences or never enough data for estimating.
- Solving the problem, using Markov Assumption:

$$P(w_1 w_2 \ldots w_n) \approx \prod_i P(w_i \mid w_{i-k} \ldots w_{i-1})$$

- Unigram:

$$P(w_1 w_2 \ldots w_n) \approx \prod_i P(w_i)$$

- Bigram: Condition on the previous word:

$$P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-1})$$

but it is not efficient because of the existence of long-distance dependencies.

- Computing N-grams:

$$P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$

Evaluating N-gram models:

1. Extrinsic (in-vivo) Evaluation: a live or real application, we put each model in a real task that we care about, run

the task, get a score for each, such as the number of words translated or transcribed correctly, and then we just compare the scores or accuracies.

Problems: Expensive, time-consuming, Doesn't always generalize to other applications

2. Intrinsic (in-vitro) Evaluation: called perplexity. Directly measures performance at predicting words. Doesn't necessarily correspond with real application performance. gives us a single general metric for language models. is useful for large language models (LLMs) as well as n-grams.
    - perplexity metric: 1. a good LM model should prefer real, or frequently observed sentences, and assign lower probably to rarely observed sentences, or just random sequences of words that we might call "word salad".
    - Shannon Game: The idea of asking an LM, or a person, to look at the start of a sentence, and predict the next upcoming word.
    - 2. intuition of perplexity is that a good LM is one that is good at the Shannon game: it assigns a higher probability to the next word that actually occurs.
    - 3. A good LM assigns high probability to the entire test set.

- the probability of a test set depends very directly on the length of the test set (probability gets smaller the longer the text) so: **Perplexity** is the inverse probability of the test set, normalized by

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

the number of words the minus is from the idea of cross-entropy rate in information theory and therefor minimizing perplexity is the same as maximizing probability and the perplexity range is [1,∞]

Chain rule: $$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

Bigrams: $$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$$

- perplexity turns out to also correspond to the weigthed average branching factor of a language. The **Branching factor of a language is the** number of possible next words that can follow any word.

Sampling:

- Sampling from a distribution: choose random points according to their likelihood.
- Sampling from a language model: generate sentence according to their likelihoods as defined by the model.
  1. Shannon: Sampling a word from a distribution.

2. Temperature sampling
3. Top-k sampling
4. Top-P sampling

- Zero probability bigrams: Things that don't ever occur in the training set, but occur in the test set so the probabilistic will be zero! This will cause problem: underestimating the probability of all sorts of words that might occur, which will hurt the performance of any application we want to run on this data. Also if the probability of any word in the test set is 0, the entire probability of the test set is 0 so we can't compute perplexity at all, since we can't divide by 0! Solution? Smoothing.

Smoothing:

- Also called Laplace smoothing
- Pretend we saw each word one more time than we did:

$$P_{Add-1}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)+1}{c(w_{i-1})+V}$$

- Adding 1 is a blunt instrument but is used where text classification or in domains where the number of zeros isn't so huge.

Backoff and Interpolation:

- helps to use **less** context(contexts you haven't learned much about)

- Backoff means use trigram if you have good evidence, otherwise bigram, otherwise unigram.
- Interpolation means mix unigram, bigram, trigram. (works better)

  Simple interpolation

$$\hat{P}(w_n|w_{n-2}w_{n-1}) = \lambda_1 P(w_n|w_{n-2}w_{n-1}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n) \qquad \sum_i \lambda_i = 1$$

  Lambdas conditional on context:

$$\hat{P}(w_n|w_{n-2}w_{n-1}) = \lambda_1(w_{n-2}^{n-1}) P(w_n|w_{n-2}w_{n-1}) + \lambda_2(w_{n-2}^{n-1}) P(w_n|w_{n-1}) + \lambda_3(w_{n-2}^{n-1}) P(w_n)$$

- called Extended Interpolated Kneser-Ney.
- Optimizing lambdas:

  Choose λs to maximize the probability of held-out data:
  - Fix the N-gram probabilities (on the training data)
  - Then search for λs that give largest probability to held-out set:

$$\log P(w_1 \ldots w_n \mid M(\lambda_1 \ldots \lambda_k)) = \sum_i \log P_{M(\lambda_1 \ldots \lambda_k)}(w_i \mid w_{i-1})$$

  - If we know all the words in advanced
    - Vocabulary V is fixed
    - Closed vocabulary task
  - Often we don't know this
    - **Out Of Vocabulary** = OOV words
    - Open vocabulary task
  - Instead: create an unknown word token <UNK>
    - Training of <UNK> probabilities
      - Create a fixed lexicon L of size V
      - At text normalization phase, any training word not in L changed to <UNK>
      - Now we train its probabilities like a normal word
    - At decoding time
      - If text input: Use UNK probabilities for any word not in training
-

- Dealing with Huge web-scale n-grams: 1. Pruning: Only store N-grams with count > threshold 2. Entropy-based pruning
- Efficiency in Huge web-scale n-grams:

  Efficiency
  - Efficient data structures like tries
  - Bloom filters: approximate language models
  - Store words as indexes, not strings
    - Use Huffman coding to fit large numbers of words into two bytes
  - Quantize probabilities (4-8 bits instead of 8-byte float)

- Smoothing for Huge web-scale n-grams: (called Stupid backoff)

$$S(w_i \mid w_{i-k+1}^{i-1}) = \begin{cases} \dfrac{\text{count}(w_{i-k+1}^{i})}{\text{count}(w_{i-k+1}^{i-1})} & \text{if } \text{count}(w_{i-k+1}^{i}) > 0 \\ 0.4S(w_i \mid w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases}$$

$$S(w_i) = \frac{\text{count}(w_i)}{N}$$

Advanced Language Modeling

- 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

$$P_{CACHE}(w \mid history) = \lambda P(w_i \mid w_{i-2} w_{i-1}) + (1-\lambda)\frac{c(w \in history)}{\mid history \mid}$$

  - These perform very poorly for speech recognition (why?)
-

Text Classification:

- *Input*: a document *d and* a fixed set of classes $C = \{c_1, c_2,\ldots, c_J\}$ - *Output*: a predicted class $c \in C$
  1. **Hand-coded rules:** Rules based on combinations of words or other features (example: spam: black-list-address OR ("dollars" AND "have been selected"))
  2. **Supervised Machine Learning:**
     - *Input:*
       - a document *d*
       - a fixed set of classes $C = \{c_1, c_2,\ldots, c_J\}$
       - A training set of *m* hand-labeled documents $(d_1,c_1),\ldots,(d_m,c_m)$
     - *Output:*
       - a learned classifier $\gamma: d \rightarrow c$

Types of Classifiers:

1. Naïve Base
2. Logistic regression
3. Support-vector machines
4. k-Nearest Neighbors

Naïve Base: Relies on very simple representation of document (called **Bag of words) =>** Just a max of a sum of weights: a linear function of the inputs So naive bayes is a linear classifier!

Bayes' Rule Applied to Documents and Classes

- For a document *d* and a class *c*

$$P(c \mid d) = \frac{P(d \mid c)P(c)}{P(d)}$$

-

"Likelihood" "Prior"

$$c_{MAP} = \underset{c \in C}{\operatorname{argmax}} P(d \mid c)P(c)$$

$$= \underset{c \in C}{\operatorname{argmax}} P(x_1, x_2, \square \ , x_n \mid c)P(c)$$

Document d represented as features x1..xn

-
- O(|X|^n•|C|) parameters
- Could only be estimated if a very, very large number of training examples was available.

Multinomial Naive Bayes Classifier

$$c_{MAP} = \underset{c \in C}{\operatorname{argmax}} P(x_1, x_2, \square \ , x_n \mid c)P(c)$$

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} P(c_j) \prod_{x \in X} P(x \mid c)$$

- By Assuming the feature probabilities $P(x_i|c_j)$ are independent given the class c. (position doesn't matter)
- Because of floating-point underflow, We'll sum logs of probabilities instead of multiplying probabilities:

$$c_{\mathrm{NB}} = \underset{c_j \in C}{\operatorname{argmax}} \left[\log P(c_j) + \sum_{i \in \mathrm{positions}} \log P(x_i|c_j) \right]$$

Learning the Multinomial Naive Bayes Model:

- maximum likelihood estimates: simply use the frequencies

$$\hat{P}(c_j) = \frac{N_{c_j}}{N_{total}}$$

$$\hat{P}(w_i \mid c_j) = \frac{count(w_i, c_j)}{\sum_{w \in V} count(w, c_j)}$$

in the data

- Create mega-document for topic *j* by concatenating all

For each c_j in C do

$docs_j \leftarrow$ all docs with class $= c_j$

$$P(c_j) \leftarrow \frac{|docs_j|}{|\text{total \# documents}|}$$

docs in this topic:

- We use smoothing for avoiding of the words which are not in the training documents:

$$\hat{P}(w_i \mid c) = \frac{count(w_i, c) + 1}{\sum_{w \in V} \left(count(w, c) + 1 \right)}$$

$$= \frac{count(w_i, c) + 1}{\left(\sum_{w \in V} count(w, c) \right) + |V|}$$

- From training corpus, extract *Vocabulary*
- **Unknown words:** Remove them from the test document! Don't include any probability for them at all! (Building an unknown word model doesn't help. knowing which class has more unknown words is not generally helpful!)

- **Stop words (the, a, …):** As removing stop words doesn't usually help, most NB algorithms use **all** words and **don't** use stopword lists. But if you want, you can Sort the vocabulary by word frequency in training set, Call the top 10 or 50 words the **stopword list**, Remove them from both training and test sets.

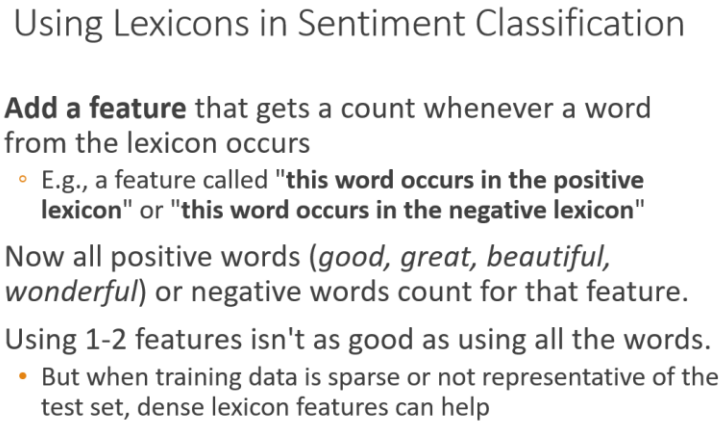
Sentiment and Binary Naive Bayes:

- For tasks like sentiment, word **occurrence** seems to be more important than word **frequency**. So we will use **Binary multinominal naive bayes**, or **binary NB!**
- Binary NB: First remove all duplicate words from document *d,* then compute NB using the same equation => here Counts can still be 2! Binarization is within-doc(within each class!)
- Dealing with negation: Add NOT_ to every word between negation and following punctuation.

```
didn't like this movie , but I
```

```
didn't NOT_like NOT_this NOT_movie but I
```

- Using lexicons(pre-built word lists) when we don't have enough labeled training data:



Naïve Bayse properties:

1. Very Fast, low storage requirements
2. Work well with very small amounts of training data
3. Robust to Irrelevant Features: Irrelevant Features cancel each other without affecting results
4. Very good in domains with many equally important features
5. Optimal if the independence assumptions hold: If assumed independence is correct, then it is the Bayes Optimal Classifier for problem.
6. A good dependable baseline for text classification

Naïve Bayes and Language Modeling:

- Naïve bayes classifiers can use any sort of feature (URL, email address, dictionaries, network features). If We use **only** word features (we use **all** of the words in the text (not

a subset)) then Naïve bayes has an important similarity to language modeling.

- Each class = a unigram language model => P(s|c) is the multiply of P(word | c).

Evaluating Classifiers:

		gold standard labels	
		gold positive	gold negative
system output labels	system positive	**true positive**	**false positive**
	system negative	**false negative**	**true negative**

- $$\textbf{accuracy} = \frac{tp+tn}{tp+fp+tn+fn}$$
-
- Although accuracy might seem a natural metric, we generally don't use it for text classification tasks. (As accuracy doesn't work well when we're dealing with uncommon or imbalanced classes)
- $$\textbf{precision} = \frac{tp}{tp+fp}$$ Precision is out of the things the system selected. (% selected items which are correct)
- $$\textbf{recall} = \frac{tp}{tp+fn}$$ Recall is out of all the correct items that should have been positive, what % of them did the system select. (% correct items which are selected)

- But again both of them are awful for uncommon or imbalanced classes.
- to get high precision, a system should be very reluctant to guess – but then it may miss somethings and have poor recall.
- to get high recall, a system should be very willing to guess but then it may return some junk and have poor precision

$$F_1 = \frac{2PR}{P+R}$$

-
- If we have more than 2 classes to compute precision of them, we can use:
    1. Macroaveraging: we compute the performance for each class, and then average over classes.
    2. Microaveraging: we first combine the decisions for all classes in a single confusion matrix, and then compute precision and recall from that table.

Harms of classification:

1. Representational Harms: Harms caused by a system that shames a social group such as by putting continuous negative image/idea about them.
2. Censorship: **Toxicity detection** is the text classification task of detecting hate speech, abuse, harassment, or other kinds of toxic language. They incorrectly flag non-toxic

sentences that simply mention minority identities (like the words "blind" or "gay") which will result in:

- Censorship of speech by disabled people and other groups.
- Speech by these groups becomes less visible online.
- Writers might be nudged by these algorithms to avoid these words making people less likely to write about themselves or these groups.

3. performance disparities: Text classifiers perform worse on many **languages** of the world due to lack of data or labels. Text classifiers perform worse on **varieties** of even high-resource languages like English. (like language identification)

These harms will cause:

1. Issues in the data; NLP systems amplify biases in training data
2. Problems in the labels
3. Problems in the algorithms

Logistic regression:

- Is the foundation of neural networks
- Logistic regression is a **discriminative** classifier (Naïve Bayse is generative classifier)
- Discriminative classifiers:
  - Building a model for each class and assigning probabilistic to each image

- For new images: Run both models and see which one fits better

Naive Bayes

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \overbrace{P(d|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}}$$

- Generative classifiers:
  - Just trying to distinguish images between different classes

Logistic Regression

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \overbrace{P(c|d)}^{\text{posterior}}$$

- Sigmoid: $$y = s(z) = \frac{1}{1+e^{-z}}$$
- $$1 - \sigma(x) = \sigma(-x)$$
- Turning probability into a classifier:

Turning a probability into a classifier

$$\hat{y} = \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases} \quad \begin{array}{l} \text{if } w \cdot x + b > 0 \\ \text{if } w \cdot x + b \leq 0 \end{array}$$

- cross-entropy loss: is the negative log likelihood loss (choose the parameters *w*,*b* that maximize the log

probability of the true *y* labels in the training data given the observations *x*)

Goal: maximize probability of the correct label $p(y|x)$

Maximize: $$\begin{aligned}\log p(y|x) &= \log\left[\hat{y}^y(1-\hat{y})^{1-y}\right] \\ &= y\log\hat{y} + (1-y)\log(1-\hat{y})\end{aligned}$$

Now flip sign to turn this into a loss: something to minimize

Cross-entropy loss (because is formula for cross-entropy(y, $\hat{y}$))

Minimize: $$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y\log\hat{y} + (1-y)\log(1-\hat{y})]$$

Or, plugging in definition of $\hat{y}$:

$$L_{\text{CE}}(\hat{y}, y) = -[y\log\sigma(w\cdot x+b) + (1-y)\log(1-\sigma(w\cdot x+b))]$$

- Optimization:

$$\hat{\theta} = \underset{\theta}{\text{argmin}} \frac{1}{m}\sum_{i=1}^{m} L_{\text{CE}}(f(x^{(i)};\theta), y^{(i)})$$

- For logistic regression, this loss function is convex which has just one minimum (but it is not in NNs and gradient descent may get stuck in local minima!)
- **Gradient Descent**: Find the gradient (a vector pointing in the direction of the greatest increase in a function) of the loss function at the current point and move in the **opposite** direction.

$$w^{t+1} = w^t - h\frac{d}{dw}L(f(x;w), y)$$

$$\frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} = [\sigma(w\cdot x+b)-y]x_j$$

$$\nabla_{w,b} = \begin{bmatrix} \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_1} \\ \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_2} \\ \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial b} \end{bmatrix} = \begin{bmatrix} (\sigma(w\cdot x+b)-y)x_1 \\ (\sigma(w\cdot x+b)-y)x_2 \\ \sigma(w\cdot x+b)-y \end{bmatrix}$$

- The learning rate η is a **hyperparameter:** too high: the learner will take big steps and overshoot, too low: the learner will take too long
- Mini-batch training: 1. computational efficiency 2. Can be be vectorized 3. Parallelism

Overfitting:

- 4-gram model on tiny data will just memorize the data and overfit so we should avoid by regularization, dropout
- Regularization:

  Add a regularization term $R(\theta)$ to the loss function
  (for now written as maximizing logprob rather than minimizing loss)

$$\hat{\theta} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(\theta)$$

- L2 regularization:

  L2 Regularization (= ridge regression)

  The sum of the squares of the weights

  The name is because this is the (square of the) **L2 norm** $||\theta||_2$, = **Euclidean distance** of θ to the origin.

$$R(\theta) = ||\theta||_2^2 = \sum_{j=1}^{n} \theta_j^2$$

  L2 regularized objective function:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} \theta_j^2$$

- L1 regularization:

  L1 Regularization (= lasso regression)

  The sum of the (absolute value of the) weights

  Named after the **L1 norm** $||W||_1$, = sum of the absolute values of the weights, = **Manhattan distance**

$$R(\theta) = ||\theta||_1 = \sum_{i=1}^{n} |\theta_i|$$

  L1 regularized objective function:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} |\theta_j|$$

Multinomial Logistic Regression:

- we need more than 2 classes
- examples: Positive/negative/neutral, Parts of speech (noun, verb, adjective, adverb, preposition, etc.), Classify emergency SMSs into different actionable classes

- using **multinomial logistic regression** like Softmax regression or Multinomial logit or Maximum entropy modeling or MaxEnt.
- Using activation function of softmax:

Turns a vector $z = [z_1, z_2, \ldots, z_k]$ of k arbitrary values into probabilities

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

The denominator $\sum_{i=1}^{k} e^{z_i}$ is used to normalize all the values into probabilities.

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

Word Meaning:

1. which the meaning of words is represented by just spelling the word with small capital letters; representing the meaning of “dog” as DOG, and “cat” as CAT. => it’s a unsatisfactory model.
2. Lemmas and senses: A sense or “concept” is the meaning component of a word. Lemmas can be polysemous (have multiple senses).
   Relation between senses:
   - Synonymy: have the same meaning in some or all contexts.
     - the word *synonym* is therefore used to describe a relationship of approximate or rough synonymy.

    - **The Linguistic Principle of Contrast:** a difference in linguistic form is always associated with some difference in meaning.
- Similarity: Words with similar meanings. Not synonyms, but sharing some element of meaning.
    - Calculating similarity can be done by asking humans to judge how similar one word is to another.
- Word relatedness (word association): Words can be related in any way, perhaps via a semantic frame or field. (sharing practically no features but are clearly related).

  One common kind of relatedness between words is if they belong to the same semantic field. Semantic field: Words that cover a particular semantic domain, bear structured relations with each other.
- Antonymy: Senses that are opposites with respect to only one feature of meaning (Otherwise, they are very similar!).

  antonyms can define a binary opposition or be at opposite ends of a scale or be *reversives*.
- *affective meanings* or connotations: the aspects of a word's meaning that are related to a writer or reader's emotions, sentiment, opinions, or evaluations.

  Words seem to vary along 3 affective dimensions:
    - **valence**: the pleasantness of the stimulus

- **arousal**: the intensity of emotion provoked by the stimulus
- **dominance**: the degree of control exerted by the stimulus

Computational models of word meaning:

- Vector semantics is the standard way to represent word meaning in NLP.
- Ludwig Wittgenstein: The meaning of a word is its use in the language, meaning its neighboring words or grammatical environments.

  Zellig Harris (1954):

  If A and B have almost identical environments we say that they are synonyms.

  Idea 1: Defining meaning by linguistic distribution

  Idea 2: Meaning as a point in multidimensional space

- :
  - Idea 1: define the meaning of a word by its distribution in language use, meaning its neighboring words or grammatical environments.
  - Idea 2: the connotation of a word is represented by 3 numbers, it's valence, arousal, and dominance. That means we are essentially representing a word's connotation by a point in three-dimensional space.

- By combining these two ideas, we will have Defining meaning as a point in space based on distribution:

  Each word = a vector (not just "good" or "w_{45}")

  Similar words are "**nearby in semantic space**"

  We build this space automatically by seeing which words are **nearby in text**

  - We define meaning of a word as a vector, called an "embedding" because it's embedded into a space (see textbook).
  - Fine-grained model of meaning for similarity.
  - Difference of words and embeddings:

    Consider sentiment analysis:

    - With **words**, a feature is a word identity
      - Feature 5: 'The previous word was "terrible"'
      - requires **exact same word** to be in training and test
    - With **embeddings**:
      - Feature is a word vector
      - 'The previous word was vector [35,22,17…]
      - Now in the test set we might see a similar vector [34,21,14]
      - We can generalize to **similar but unseen** words!!!

- Two kinds of embeddings and their difference:

  tf-idf
  - Information Retrieval workhorse!
  - A common baseline model
  - **Sparse** vectors
  - Words are represented by (a simple function of) the **counts** of nearby words

  Word2vec
  - **Dense** vectors
  - Representation is created by training a classifier to **predict** whether a word is likely to appear nearby
  - Later we'll discuss extensions called **contextual embeddings**

term-document matrix:

1. has $|V|$ rows (one for each word type in the vocabulary) and D columns, one for each document in the collection:

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

2. vector semantics:
   - for representing the meaning of *words.*
   - associating each word with a word vector— a row vector rather than a column vector.
   - similar words have similar vectors because they tend to occur in similar documents.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

- Two **words** are similar in meaning if their context vectors are similar.

term-context matrix (word-word matrix or the term-term matrix):

- Two **words** are similar in meaning if their context vectors are similar.
- the columns are labeled by words.
- dimensionality $|V| \times |V|$ and each cell records the number of times the row (target) word and the column (context) word co-occur in some context in some training corpus.

word similarity:

- The Raw dot product acts as a similarity metric because it will tend to be high just when the two vectors have large values in the same dimensions:

$$\text{dot product}(\mathbf{v}, \mathbf{w}) = \mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \ldots + v_N w_N$$

  - Problems: favors long vectors => Frequent words (of, the, you) have long vectors (since they occur many times with other words).
  - The dot product is higher if a vector is longer.
  - More frequent words have longer vectors, since they tend to co-occur with more words and have higher co-occurrence values with each of them.

$$|\mathbf{v}| = \sqrt{\sum_{i=1}^{N} v_i^2}$$

- Vector length:
-

$$\text{cosine}(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

- Using cosine:
  - Ranges from -1 to 1: 1 for similar vectors and -1 for dissimilar vectors and 0 is for orthogonal. => since raw frequency values are non-negative, the cosine for term-term matrix vectors ranges from 0–1.
  - the cosine is larger but the angle is smaller.

word weighting problem:

- The co-occurrence matrices we have seen represent each cell by word frequencies. Frequency is clearly useful; if *sugar* appears a lot near *apricot*, that's useful information. But overly frequent words like *the*, *it,* or *they* are not very informative about the context
- Solutions:

tf-idf: tf-idf value for word t in document d:

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

1. Tf-idf: Words like "the" or "it" have very low idf

- use of a special weight call the "inverse document frequency”.
- Tf: frequency of word t in the document d => but instead we use this formula:

$$\mathrm{tf}_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{count}(t,d) & \text{if } \mathrm{count}(t,d) > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Df(document frequency):
  - df_t *is* the number of documents *t* occurs in.
  - if the words have identical collection frequencies but very different document frequencies, If our goal is to find documents about the romantic tribulations of Romeo(less df), the word *Romeo* should be highly weighted, but not *action*.
- idf:
  - $$\mathrm{idf}_t = \log_{10}\left(\frac{N}{\mathrm{df}_t}\right)$$ log(all of documents / number of documents contain term t)
  - give a higher weight to words that occur only in a few documents.
  - The document frequency idf of a term *t* is the number of documents it occurs in.

- The lowest weight of 1 is assigned to terms that occur in all the documents.

2. PMI:

PMI: (Pointwise mutual information)

- $\text{PMI}(w_1, w_2) = log \frac{p(w_1,w_2)}{p(w_1)p(w_2)}$

See if words like "good" appear more often with "great" than we would expect by chance

- compares the probabilities we see with what we would have expected by change.

Word2vec:

- Dense vectors may **generalize** better than explicit counts (the smaller parameter space possibly helps with generalization)
- Dense vectors may do better at capturing synonymy.
- The word2vec methods are fast, efficient to train, and easily available online with code and pretrained embeddings.
- Word2vec embeddings are static embeddings, meaning that the method learns one fixed embedding for each word in the vocabulary.
- Train a classifier on a binary **prediction** task and then take the learned classifier weights as the word embeddings

skip-gram:

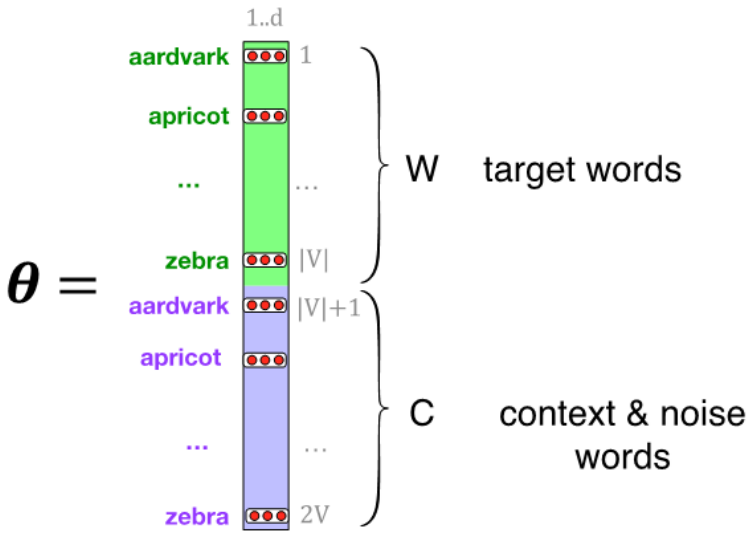
Approach: predict if candidate word *c* is a "neighbor"

1. Treat the target word *t* and a neighboring context word *c* as **positive examples**.
2. Randomly sample other words in the lexicon to get negative examples
3. Use logistic regression to train a classifier to distinguish those two cases
4. Use the learned weights as the embeddings

-
- Goal: train a classifier that is given a candidate (**w**ord, **c**ontext) pair.
- To compute similarity between these dense embeddings, we rely on the intuition that two vectors are similar if they have a high dot product.
- Skip-Gram Classifier computes $P(+|w, c)$:

$$P(+|w, c_{1:L}) = \prod_{i=1}^{L} \sigma(c_i \cdot w)$$

$$\log P(+|w, c_{1:L}) = \sum_{i=1}^{L} \log \sigma(c_i \cdot w)$$

- 

context. Thus the parameters we need to learn are two matrices *W* and *C*, each containing an embedding for every one of the |*V* | words in the vocabulary.
positive examples +		negative examples -			
t	c	t	c	t	c
apricot	tablespoon	apricot	aardvark	apricot	seven
apricot	of	apricot	my	apricot	forever
apricot	jam	apricot	where	apricot	dear
apricot	a	apricot	coaxial	apricot	if
- Given the set of positive and negative training instances, and an initial set of embedding vectors

  The goal of learning is to adjust those word vectors such that we:
  - **Maximize** the similarity of the target word, context word pairs (w , c_{pos}) drawn from the positive data
  - **Minimize** the similarity of the (w , c_{neg}) pairs drawn from the negative data.
-

- Loss function:

$$
\begin{aligned}
L_{CE} &= -\log\left[P(+|w,c_{pos})\prod_{i=1}^{k}P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos}) + \sum_{i=1}^{k}\log P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos}) + \sum_{i=1}^{k}\log\left(1-P(+|w,c_{neg_i})\right)\right] \\
&= -\left[\log \sigma(c_{pos}\cdot w) + \sum_{i=1}^{k}\log\sigma(-c_{neg_i}\cdot w)\right]
\end{aligned}
$$

- Gradient descent: The skip-gram model tries to shift embeddings so the target embeddings are closer to (have a higher dot product with) context embeddings for nearby words and further from (lower dot product with) context embeddings for noise words that don't occur nearby.

$$
\begin{aligned}
\frac{\partial L_{CE}}{\partial c_{pos}} &= [\sigma(c_{pos}\cdot w)-1]w \\
\frac{\partial L_{CE}}{\partial c_{neg}} &= [\sigma(c_{neg}\cdot w)]w \\
\frac{\partial L_{CE}}{\partial w} &= [\sigma(c_{pos}\cdot w)-1]c_{pos} + \sum_{i=1}^{k}[\sigma(c_{\mathrm{neg}_i}\cdot w)]c_{neg_i}
\end{aligned}
$$

-

$$
\begin{aligned}
c_{pos}^{t+1} &= c_{pos}^{t} - \eta[\sigma(c_{pos}^{t}\cdot w^{t})-1]w^{t} \\
c_{neg}^{t+1} &= c_{neg}^{t} - \eta[\sigma(c_{neg}^{t}\cdot w^{t})]w^{t} \\
w^{t+1} &= w^{t} - \eta\left[[\sigma(c_{pos}\cdot w^{t})-1]c_{pos} + \sum_{i=1}^{k}[\sigma(c_{\mathrm{neg}_i}\cdot w^{t})]c_{neg_i}\right]
\end{aligned}
$$

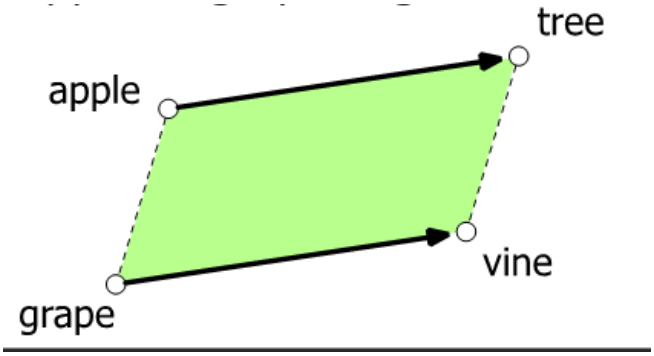
-

- SGNS learns two sets of embeddings: Target embeddings matrix W and Context embedding matrix C. It's common to

just add them together, representing word *i* as the vector $w_i + c_i$

Properties of Embeddings:

- Size of window:
  - Shorter context windows tend to lead to representations that are a bit more syntactic, since the information is coming from immediately nearby words.
  - When vectors are computed from long context windows, the highest cosine words to a target word *w* tend to be words that are topically related but not similar. => nearest words are related words in same semantic field.
- Another semantic property of embeddings is their ability to capture relational meanings:



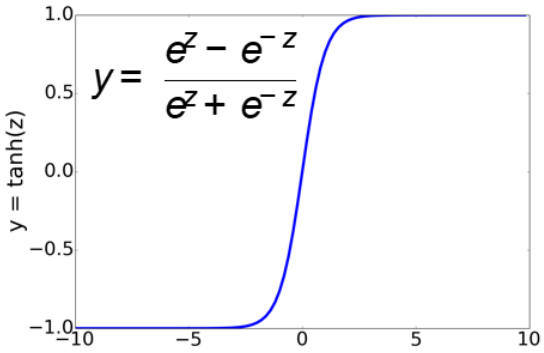
For a problem a:a*::b:b*, the parallelogram method is:

$$\hat{b}^* = \underset{x}{\text{argmax}}\ \text{distance}(x, a^* - a + b)$$

This parallelogram method only seems to work for frequent

words, small distances and certain relations (relating countries to capitals, or parts of speech), but not others

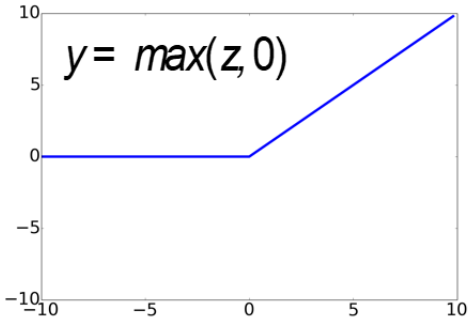
NN:



tanh

-
- **vanishing gradient problem:** In the sigmoid or tanh functions, very high values of *z* result in values of *y* that are **saturated**, i.e., extremely close to 1 and have derivatives very close to 0.
- Solving the problem of vanishing by RELU:

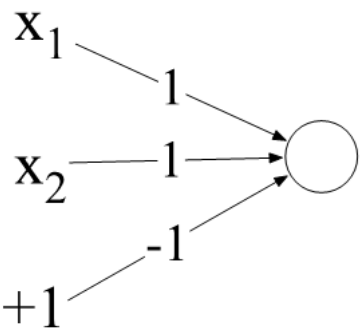
Most Common:



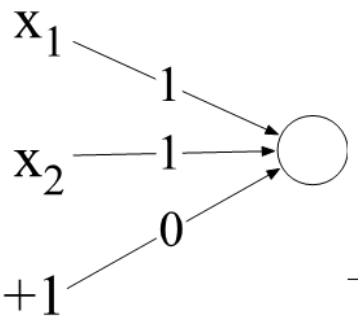
ReLU

- Perceptron: that has a binary output and does not have a non-linear activation function:

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



  - AND:



  - OR:
  - Perceptrons are linear classifiers. So we cannot implement XOR:

$$w_1 x_1 + w_2 x_2 + b = 0$$

(in standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$)

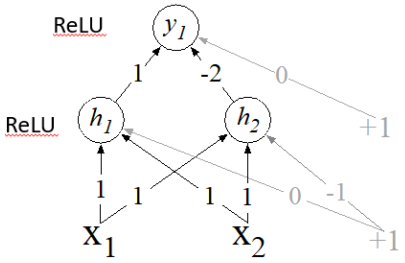
This line acts as a **decision boundary**

- 0 if input is on one side of the line
- 1 if on the other side of the line

XOR **can't** be calculated by a single perceptron

XOR **can** be calculated by a layered network of units.

XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



- MLP:

Replacing the bias unit

Let's switch to a notation without the bias unit

Just a notational change

1. Add a dummy node $a_0=1$ to each layer
2. Its weight w_0 will be the bias
3. So input layer $a^{[0]}_0=1$,
   - And $a^{[1]}_0=1$, $a^{[2]}_0=1$,…

Just adding a hidden layer to logistic regression

- allows the network to use non-linear interactions between features
- which may (or may not) improve performance.

Even better: representation learning

The real power of deep learning comes from the ability to **learn** features from the data

Neural Net Classification with embeddings as input features:

- The input size is not always the same, so:

  Some simple solutions (more sophisticated solutions later)

  1. Make the input the length of the longest review
     - If shorter then pad with zero embeddings
     - Truncate if you get longer reviews at test time
  2. Create a single "sentence embedding" (the same dimensionality as a word) to represent all the words
     - Take the mean of all the word embeddings
     - Take the element-wise max of all the word embeddings
       - For each dimension, pick the max value from all words

Neural Language Models (LMs):

- Calculating the probability of the next word in a sequence given some history.
- predict next word w_t , given prior words w_{t-1}, w_{t-2}, w_{t-3}, …

  Why Neural LMs work better than N-gram LMs

  Training data:

  We've seen: I have to make sure that the cat gets fed.

  Never seen: dog gets fed

  Test data:

  I forgot to make sure that the dog gets ___

  N-gram LM can't predict "fed"!

  Neural LM can use similarity of "cat" and "dog" embeddings to generalize and predict "fed" after dog

-

Computation Graphs:

- For training, we need the derivative of the loss with respect to each weight in every layer of the network but the loss is computed only at the very end of the network! Solution: **error backpropagation**

Backprop is a special case of **backward differentiation**

Which relies on **computation graphs**.

- $$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}$$

- $$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

- Cross-entropy:

$$L(\hat{y}, y) = -(\mathrm{y}\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

$\hat{y} = a^{[2]}$

$$L(a, y) = -(\mathrm{y}\log a + (1-y)\log(1-a))$$

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}$$

$$\frac{\partial L}{\partial a} = -\left(\left(y\frac{\partial \log(a)}{\partial a}\right) + (1-\mathrm{y})\frac{\partial \log(1-a)}{\partial a}\right)$$

$$= -\left(\left(y\frac{1}{a}\right) + (1-\mathrm{y})\frac{1}{1-a}(-1)\right) = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)$$

$$\frac{\partial a}{\partial z} = a(1-a) \qquad \frac{\partial L}{\partial z} = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)a(1-a) = \mathrm{a} - \mathrm{y}$$

Parts of Speech:

Closed class words

- Relatively fixed membership
- Usually **function** words: short, frequent words with grammatical function
  - determiners: ***a, an, the***
  - pronouns: ***she, he, I***
  - prepositions: ***on, under, over, near, by, …***

Open class words

- Usually **content** words: Nouns, Verbs, Adjectives, Adverbs
  - Plus interjections: **oh, ouch, uh-huh, yes, hello**
- New nouns and verbs like *iPhone* or *to fax*

- Closed class words are generally function words like *of*, *it*, *and*, or *you*, which tend to be very short, occur frequently, and often have structuring uses in grammar.

- Closed classes are those with relatively fixed membership, such as prepositions.
- new nouns and verbs like *iPhone* or *to fax* are continually being created or borrowed.
- Verbs refer to actions and processes, including open-class. closed-class contains auxiliary verbs.
- conjunctions that join two phrases, clauses, or sentences are closed-class

Why Part of Speech Tagging?

- Can be useful for other NLP tasks
  - Parsing: POS tagging can improve syntactic parsing
  - MT: reordering of adjectives and nouns (say from Spanish to English)
  - Sentiment or affective tasks: may want to distinguish adjectives or other POS
  - Text-to-speech (how do we pronounce "lead" or "object"?)
- Or linguistic or language-analytic computational tasks
  - Need to control for POS when studying linguistic change like creation of new words, or meaning shift
  - Or control for POS in measuring meaning similarity or difference

- About 97% accuracy hasn't changed in the last 10+ years. HMMs, CRFs, BERT perform similarly. Human accuracy about the same.
- baseline is 92%. Baseline is performance of stupidest possible method.

Sources of information for POS tagging

Janet will back the bill
AUX/NOUN/VERB? NOUN/VERB?

Prior probabilities of word/tag
- "will" is usually an AUX

Identity of neighboring words
- "the" means the next word is probably not a verb

Morphology and wordshape:
- Prefixes unable: un- → ADJ
- Suffixes importantly: -ly → ADJ
- Capitalization Janet: CAP → PROPN

Named Entity Recognition (NER):

- **Named entity**, in its core usage, means anything that can be referred to with a proper name. Most common 4 tags:
  - PER (Person): "Marie Curie"
  - LOC (Location): "New York City"
  - ORG (Organization): "Stanford University"
  - GPE (Geo-Political Entity): "Boulder, Colorado"
- Often multi-word phrases
- But the term is also extended to things that aren't entities:
  - dates, times, prices

- being a proper noun is a grammatical property of these words. But viewed from a semantic perspective, these proper nouns refer to different kinds of entities.
- A named entity is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization.

Why NER?

Sentiment analysis: consumer's sentiment toward a particular company or person?

Question Answering: answer questions about an entity?

Information Extraction: Extracting facts about entities from text.

-

Why NER is hard

1) Segmentation
   - In POS tagging, no segmentation problem since each word gets one tag.
   - In NER we have to find and segment the entities!
2) Type ambiguity

-
- The standard approach to sequence labeling for a span-recognition problem like NER is BIO tagging. This is a method that allows us to treat NER like a word-by-word sequence labeling task, via tags that capture both the boundary and the named entity type.

BIO Tagging

B: token that *begins* a span

I: tokens *inside* a span

O: tokens outside of any span

-

- The number of tags is thus 2*n* + 1 tags, where *n* is the number of entity types:

Words	BIO Label
Jane	B-PER
Villanueva	I-PER
of	O
United	B-ORG
Airlines	I-ORG
Holding	I-ORG
discussed	O
the	O
Chicago	B-LOC
route	O
.	O

- BIO Tagging variants: IO(there is not any B-tag so begin is also inside) and BIOES(begin, inside, outside, end, single):

"Barack Obama visited New York City last week."

Here's how each word in the sentence would be tagged using the BIOES format:

- **Barack**: B-PER (Begin-Person)
- **Obama**: E-PER (End-Person)
- **visited**: O (Outside any named entity)
- **New**: B-LOC (Begin-Location)
- **York**: I-LOC (Inside-Location)
- **City**: E-LOC (End-Location)
- **last**: O (Outside any named entity)
- **week**: O (Outside any named entity)