COMPSCI4074 / COMPSCI5096 Text as Data H / M - 2023-24 - Wiki for making course notes

COMPSCI4074 / COMPSCI5096 Text as Data H / M - 2023-24 - Course notes

Table of contents

Week 1 - introduction, tokens and document similarity [edit]

Introduction

- Structured data = like tables or databases, with column headings providing labels on the meaning of each cell
- Unstructured data = section of text with no labels to provide meaning

Text encodings: ASCII and Unicode

- ASCII = 128 possible characters in one byte (A-Z, a-z, 0-9, punctuation, control characters e.g. newline)
- Unicode = variable-byte (2-4 bytes) encoding
  - encodes alphabets of many languages
  - emojis, maths symbols, and more
  - frequently updated with new characters
- Being unaware of Unicode can cause odd character errors ("mojibake")

Document similarity

- Data cleaning often necessary: may need to remove bold, italics, URLs (if not useful), tables (or just their formatting)
- More overlap = higher similarity hypothetically (talking about similar subject)
- Consider three possible ways of measuring overlap between two documents/posts:
  1. Individual letters
  2. Short sequences of letters
  3. Words

Using individual letters

- Not a good measure of overlap, not typically used for text analysis
- Does not consider order and grouping of letters, which is key
  - not enough information for majority of uses
- It is possible to exploit the fact that different letters occur at different frequencies to detect other languages (but this is a suboptimal method; using pair/triple frequencies (e.g. n-grams) can provide more data)

Using sequences of letters (a.k.a. character n-grams)

- Some sequences can be quite specific (e.g. "irn"), making this method more useful than individual letters
  - has problems, but can be very efficient
- Bi-grams (n=2) are pairs of neighbouring characters (e.g. "ir", "rn", "n ", " b", "br", "ru")
- Tri-grams (n=3) are triples of neighbouring characters (e.g. "irn", "rn ", "r b", " br", "bru")
- n-grams can be extracted with a sliding window of length n moving across the text

Strengths and weaknesses of character n-grams

- Strengths:
  - more specific than individual letters (e.g. tri-grams can capture short words)
  - easy to implement
- Weaknesses:
  - using small n (e.g. bigrams) won't be specific enough for many words
  - large n may create n-grams that are very rare
  - same n-gram can come from different words with different meaning (e.g. "ong" is in "wrong" and "strong")

Using words

- To split up a section of text into "words", we actually split it up into individual tokens
  - tokenisation captures meaning better than character n-grams, word n-grams could be used to capture extra information
- Token = technical name for meaningful sequence of characters
- A token is NOT a word
  - words can be broken into separate tokens (e.g. "don't" -> "do", "n't")
  - tokens can be punctuation, numbers, some whitespace, other spans of text
- Languages other than English can have very different tokenisation challenges

Tokenising with handmade rules

- Example sentence: "John's father didn't have £100."
- Can split with whitespace (spaces, tabs, etc.)

John’s	father	didn’t	have	£100.

- Can split with whitespace + punctuation (,.;!@#$%, etc)

John	‘	s	father	didn	‘	t	have	£	100	.

- Ideal tokenisation (requires handcrafted rules)

John	‘s	father	did	n’t	have	£	100	.

Tokenising with spaCy

- Good tokenisation requires a lot of handcrafted rules
- spaCy implements these for us

Filtering using stopwords

- Some words are more important than others for representing overall meaning
- Some common words convey minimal meaning - a common practice is to remove these
- These words are called "stopwords" - there are many lists of stopwords but none are definitive
- BUT some important phrases contain common words (e.g. "The Who", "to be or not to be")

Two words only map together if they match exactly!

- Two sentences don't overlap at all if only considering individual words, e.g.
  "I loved IRN BRU", "He loves irn bru"
- Common to make the text case insensitive ("IRN" and "irn") by converting all text to lowercase
  - but you may lose some important info provided by the case
- We need to remove suffixes to account for word forms ("loved", "loves"), reducing words to their canonical versions (e.g. loved -> love)
  - but you may lose some meaning

Breaking words into stems and affixes

- Stems = core meaning-bearing units (e.g. "love" is the stem of "loved")
- Affixes = bits/pieces that adhere to stems (e.g. "-d" in "loved" or "-s" in "eggs")
  - often affixes are grammatical additions for conjugation/tense
- Stems and affixes are broadly known as "morphemes"
- Morpheme = small meaningful units that make up words
- We need to do stemming as we want the stems

Stemming

- Stemming = process for reducing inflected words to their stem or root form
- Often rule-based, and removes common suffixes (-d, -ing, -s, etc.)
- Generally works on single words with no context
- Can make mistakes but often these aren't bad (e.g. computing -> comput)

The Porter Stemmer (popular stemming algorithm)

Step 1a

```
sses → ss   caresses → caress
ies  → i    ponies   → poni
ss   → ss   caress   → caress
s    → ø    cats     → cat
```

Step 1b

```
(*v*)ing → ø  walking   → walk
              sing      → sing
(*v*)ed  → ø  plastered → plaster
…
```

Step 2 (for long stems)

```
ational→ ate relational→ relate
izer→ ize    digitizer → digitize
ator→ ate    operator  → operate
…
```

Step 3 (for longer stems)

```
al    → ø  revival    → reviv
able  → ø  adjustable → adjust
ate   → ø  activate   → activ
…
```

Lemmas

- Two words have the same lemma if they have the:
  - same stem
  - same part-of-speech (e.g. noun, verb, etc.)
  - same essential meaning
- Difference between lemmas and stems:
  - lemmas take in the context of the sentence

Use word n-grams to capture additional meaning

- Words are not independent - neighbouring words can change their meaning (e.g. "really **like**" and "don't **like**")
- We can capture this with pairs/triples of **words**
  - can use a sliding window of the pairs/triples
- This is **different** to **character** n-grams!
- Word bi-grams could capture "irn bru" as two words instead of "iru" and "bru" individually
- Word bi-grams/tri-grams are fairly powerful, but can be computationally expensive

Document similarity using set similarity

Comparing documents with sets of tokens

- Set = a group of unique items
- A document can be represented as the set of the (possibly stemmed) tokens within it
- Some characteristics of sets:
  - no concept of item frequency (words that occur many times are treated the same as those that appear once)
  - very efficient computation to compare sets
  - sets are a standard data structure in Python and most other languages

Set operations

- | X | = number of elements in set X
- X ∩ Y = intersection of sets X and Y (the items that are in both X and Y)
- X ∪ Y = union of sets X and Y (the items that are in either X or Y)
- X - Y = set difference of X and Y (the items that are in X but not Y)

Set similarity measures

- sim(X, Y) = function that calculates a similarity measure for two sets of tokens (X and Y) for two documents to compare
- sim(X, Y) should be high when X and Y are similar
- sim(X, Y) should be low when X and Y are dissimilar

Number of overlapping words

- **sim(X, Y) = |X ∩ Y|**
- Can argue two documents are similar if they contain many of the same words
- But we need a normalised (between 0 and 1) value
  - a long document that contains a large selection of words would match with lots of documents, so need to factor in the number of tokens in each token set

Overlap Coefficient

- **sim(X, Y) = |X ∩ Y| / min(|X|, |Y|)**
- This means the % of unique tokens in the smaller document that appear in the larger document
- Why this is better than raw number of overlapping words:
  - using the number of unique tokens in X and Y to normalise the measure
  - is bounded between 0 (no overlapping tokens) and 1 (one set is a subset of the other)

Sørensen-Dice Coefficient

- **sim(X, Y) = (2 |X ∩ Y|) / |X| + |Y|**
- Properties: bounded between 0 (no overlap) and 1 (perfect overlap)
- Difference with Overlap Coefficient:
  - will only be 1 if two sets are exactly matching, not if one set is a subset of the other

Jaccard Similarity

- **sim(X, Y) = |X ∩ Y| / |X ∪ Y|**
- Means the number of overlapping tokens divided by the number of unique tokens across the two documents
- Properties: bounded between 0 and 1 (like Sørensen-Dice), very popular
- Difference with Sørensen-Dice Coefficient:
  - Jaccard distance (1 - sim(X, Y) satisfies the triangle inequality such that:
  dist(X, Z) <= dist(X, Y) + dist(Y, Z)
  - this makes Jaccard a metric
  - Sørensen-Dice doesn't satisfy this, so it is only a semi-metric

Properties of a metric

- Distance between X and X is zero: d(X, X) = 0
- Positive: d(X, Y) > 0 where X != Y
- Symmetric: d(X, Y) = d(Y, X)

- Triangle inequality holds: dist(X, Z) <= dist(X, Y) + dist(Y, Z)

Week 2 - geometric similarity, text distributions, and clustering

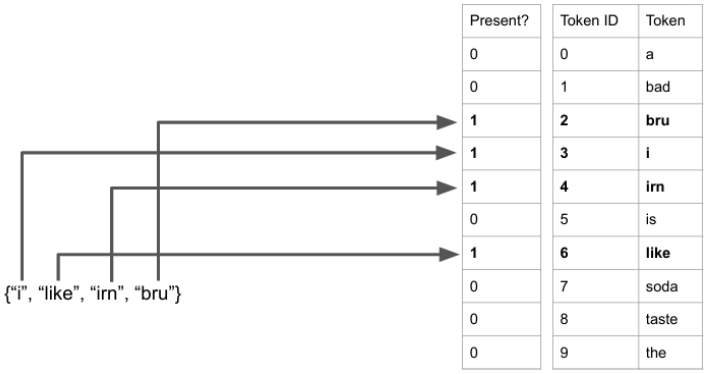
[edit]

Going from sets to sparse vectors

Treating a set as a vector

Token ID	Token
0	a
1	bad
2	bru
3	i
4	irn
5	is
6	like
7	soda
8	taste
9	the

- Step 1: build a vocabulary from all tokens **in the corpus** and assign each token an integer ID



Present?	Token ID	Token
0	0	a
0	1	bad
1	**2**	**bru**
1	**3**	**i**
1	**4**	**irn**
0	5	is
1	**6**	**like**
0	7	soda
0	8	taste
0	9	the

- Step 2: build a vector (the "Present?" column above) that is all zeroes EXCEPT at the indices of the words **in the set**, and is the same length as the vocabulary size

- In practice, most of the values in the vector will be 0
- vocabulary size >> document length
- Since most of the values are 0, we call this sparse **data**
- We could allocate an entire vector for the data (e.g. as an array) - this is **dense representation** which is **fixed length, vocabulary size**:
  {"i", "like", "irn", "bru"} = [0, 0, 1, 1, 1, 0, 1, 0, 0, 0, ...]
- Drawback: lots of wasted memory storing 0s

- It is often better to store sparse **data** as a **sparse representation**
- Store pairs of (token id, value); value is always 1 for now, *see term frequency*
- e.g.
  {"i", "like", "irn", "bru"} = [(2, 1), (3 ,1), (4, 1), (6, 1)]
- This is **variable length (document length)**

Sparse data vs sparse representation

- Your data (e.g. document vector) can be either sparse or dense
- If you have **sparse (mostly zero) data** use **sparse representation (id-value pairs)**
- If you have **dense (mostly non-zero) data** use **dense representation (fixed-length vocabulary sized array)**

Vectors for an entire corpus

- You can represent an entire corpus as a document-term matrix (DTM)
- Each row is a document, each column is a term
- Each document is represented by a vector of its word occurrences

$$Doc_1 = (t_1, t_2, t_3, \ldots t_{|V|})$$
$$Doc_2 = (t_1, t_2, t_3, \ldots t_{|V|})$$

- $$\begin{array}{lcccc} & Term_1 & Term_2 & \ldots & Term_t \\ Doc_1 & d_{11} & d_{12} & \ldots & d_{1t} \\ Doc_2 & d_{21} & d_{22} & \ldots & d_{2t} \\ \vdots & \vdots & & & \\ Doc_n & d_{n1} & d_{n2} & \ldots & d_{nt} \end{array}$$

- Alternatively, sparse versions can consist of lists of ID-value pairs (postings), called an index; there are two common orientations
- Direct index = document-oriented, compressed sparse column (CSC)
  doc1 = [(TokenID, value), (TokenID, value), ...]
  doc2 = [(TokenID, value), (TokenID, value), ...]
  ...
- Inverted index = token-oriented, compressed sparse row (CSR)
  term1 = [(DocID, value), (DocID, value), ...]
  term2 = [(DocID, value), (DocID, value), ...]
  term3 = [(DocID, value), (DocID, value), ...]
  ...

Bag-of-Words model - beyond one-hot encoding

Importance

- A simple heuristic for importance is that tokens that appear a lot in one document, but don't appear a lot in most other documents are important

UNK tokens

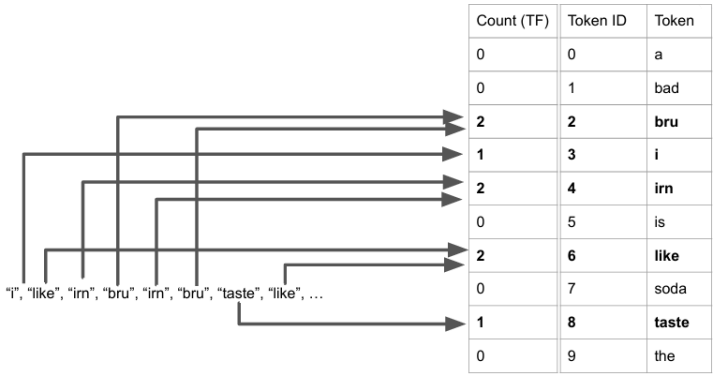
- With Bag-of-Words, you need a pre-defined vocabulary
- You need an UNK token for when you run into unknown new tokens

Dealing with punctuation

- Often you should not include punctuation in the vocab for BoW (depends on applications)
- "???" in a document could convey some meaning but "." is likely not informative
- Sometimes punctuation is inside tokens (e.g. n't)

Term frequency

- Assumption: if a term occurs a lot in a document, it should imply something about what that document is about
- This is a relaxation of the binary occurrence assumption
- Recording the term frequency information provides more information of "aboutness"
- We can make a term frequency (TF) vector, counting the number of occurrences of each term appearing in a document



Count (TF)	Token ID	Token
0	0	a
0	1	bad
2	**2**	**bru**
1	**3**	**i**
2	**4**	**irn**
0	5	is
2	**6**	**like**
0	7	soda
1	**8**	**taste**
0	9	the

- In memory this can be:
  dense rep: [0, 0, 2, 1, 2, 0, 2, 0, 1, 0, ...]
  sparse rep (token id, count): [(2, 2), (3, 1), (4, 2), (6, 2), (8, 1)]

Problem with term frequency

- Important words are usually common in a document, but common words are not necessarily important
- Articles/pronouns (e.g. I, a, the, it, you, your)
  - essentially all documents in English use these, we can delete with stopword list
- Words that are frequent but do not convey much information (e.g. "BBC" in a corpus of news articles, "study" in a corpus of scientific papers)
- Co-relations between words (e.g. novak and djokovic occur much more frequently together rather than separately)
- Magnitudes of term frequencies (e.g. if we have 4 "djokovic" occurrences, does this mean a document is twice as much about "visa" (2 occurrences))?

Problem 1: raw term frequency (TF)

- Let $tf_{t,d}$ denote the number of occurrences of term t in document d
- Raw TF has a problem: a document with 10 occurrences of a term is more related to the term than a document with 1 occurrence
  - but it is not 10 times more relevant
- Aboutness does not increase linearly with term frequency
- We can solve this with sublinear TF scaling:
  - the log frequency weight of term t in d is $w_{t,d}$
- $\mathbf{w_{t,d} = 1 + \log_{10} tf_{t,d}}$ **(if** $\mathbf{tf_{t,d} > 0}$**), otherwise 0**
- This impacts very large term frequencies (e.g. 1000 raw TF reduces to 4 log frequency weight, 10 raw TF reduces to 2 log frequency weight)

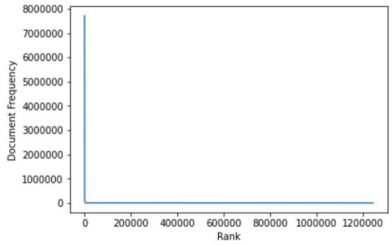
Problem 2: how "good" is each term?

- Words that occur in many documents do not provide much new "information"
- Common terms (the, of, etc.) not very helpful
- Rare terms (misspellings, etc.) sometimes also not informative, sometimes these are removed entirely (pruned)
- We can measure how many documents a term appears in to predict how useful it is (this is called document frequency)
- Terms that occur in more documents are **less** useful than rare terms
- A higher document frequency indicates a term is **less** important
- Therefore, we want the inverse document frequency (IDF)
  - a higher inverse document frequency = term is more important
- $\mathbf{idf_t = \log(N / df_t)}$
- Normally in base 10, but base 2 more common in practice

TF vs. DF vs. IDF

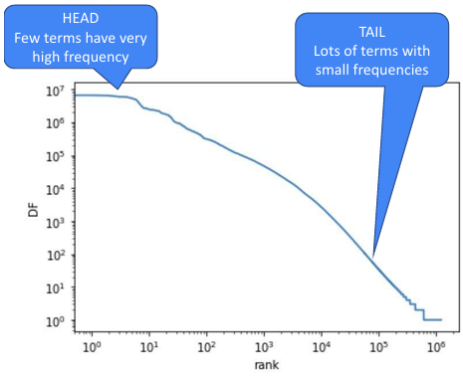
- High term frequency = document might be about this term
- High document frequency = term is probably uninformative
- High inverse document frequency = term is probably informative
- Low term frequency = document probably isn't about this term
- Low document frequency = term is probably informative
- Low inverse document frequency = term is probably uninformative

Plotting rank vs. document frequency

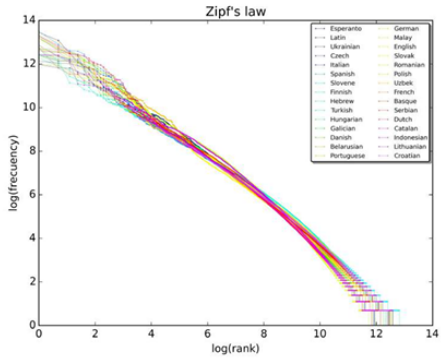


- If we plotted document frequency with a large number of tokens, from a large number of documents, with tokens ranked numerically, it would be very difficult to interpret

Plotting rank vs. document frequency with logarithmic scales (Zipf's law)



- The most frequent terms are not very useful (but there are only a few of them)
- There are lots of infrequent terms that can be useful



- Zipf's law states that word frequency is inversely proportional to word rank
  - it shows why IDF is useful and why we apply a logarithm to DF
- Zipf's law is a statistical property of language, that can be approximated as a straight-line in log-scale
- Zipf's law is an example of a "power law" - many of these laws occur in nature

TF-IDF weighting

- The TF-IDF weight of a term is the product of its TF weight and its IDF weight
- $w_{t,d} = (1 + \log(tf_{t,d})) * \log(N / df_t)$
- This is a strong weighting scheme for text similarity
- Increases with the number of occurrences within a document
- Increases with the rarity of a term in the collection
- We often use TF-IDF weights instead of term frequency counts

TF-IDF variants

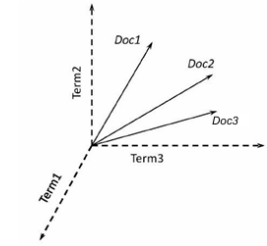
Term frequency		Document frequency		Normalization	
n (natural)	$tf_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(tf_{t,d})$	t (idf)	$\log \frac{N}{df_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+...+w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-df_t}{df_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1+\log(tf_{t,d})}{1+\log(ave_{t \in d}(tf_{t,d}))}$				

Informative or discriminative

- We can examine the collection as a whole for information about how informative or discriminative each term is

Geometric similarity

Vector space model



- We can plot vectors for each document
- Documents that are "close together" in vector space "talk about" the same things, i.e. they are similar
- 3D pictures are useful, but can be misleading for high-dimensional space

Euclidean distance

- **|| D_1 - D_2 ||**
- Measures the distance between the endpoints of two vectors
- Problem: this distance is large for vectors of different lengths

Vector space similarity basics

- Similarity = distance between points representing units of texts
- Similarity is more common than a distance or dissimilarity measure
- Core geometry formulae:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos\theta$$

$$\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} = \cos\theta$$

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

- |v| is the square root of the (sum of the squares of each element of v)

Dot product

- The dot product of two equal-length term vectors D1 and D2 is a scalar
- It is the sum of the products of each element in the equivalent position of both vectors:

$$dot - product(D_1, D_2) = \overrightarrow{D_1} * \overrightarrow{D_2} = \sum_{i=1}^{|V|} v_i w_i = v_1 * w_1 + v_2 * w_2 + \ ... + v_{|V|} * w_{|V|}$$

- Example: (1, 2, 3) · (2, 3, 4) = (1 * 2) + (2 * 2) + (3 * 4) = 20

Cosine similarity

- Uses the angle (instead of the distance) between two vectors
- Benefit: we ignore vector magnitude, so long/short documents can be similar to each other
- **cosine(D1, D2) = (D1 · D2) / |D1| |D2|**
- where **D1 · D2** is the dot product of the two vectors, and **|D1| |D2|** is the product of their magnitudes
- Importantly this transforms the angle (we want a high score for similar documents, identical documents would have a raw angle difference of 0!)

Clustering

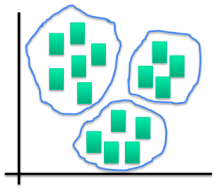
Motivation for clustering

- We want to group sets of items by similarity (e.g. group products based on features, news articles/tweets based on topic)

Purpose of clustering

- Discover the underlying structure of data by grouping similar objects (text documents) together
- Cluster documents by similarity of contained terms
- Documents within the same cluster are assumed to be similar
- We can also cluster terms (e.g. terms that often co-occur in the same documents)

How many clusters?



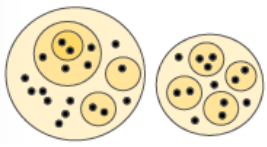
- Documents often represented by VERY high dimensional vectors, 2D is used for simplicity here
- Often there is no definitive "right" number of clusters
- The number of clusters and how they are defined vary depending on the clustering algorithm used
- Using different clustering algorithms = different clusters
- Clustering is (often) an unsupervised learning task
- Difference from supervised classification: the clusters (classes) are not pre-defined
  - no labelled training data
  - clusters are discovered by the algorithm

The clustering process

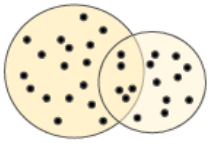
1. Derive a document representation (typically vectors of weighted terms)
2. Measure similarity between documents
3. Apply a clustering method
4. Check the validity/quality of the clustering

Clustering methods

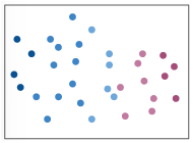
Key properties of clustering algorithms



- Partitioning criteria
  - single level vs. multi-level hierarchical partitioning
  - multi-level is often desirable

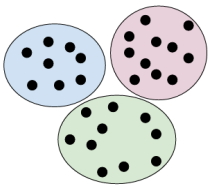


- Separation of clusters
  - exclusive = one object belongs to only one cluster
  - overlapping = one object may belong to multiple clusters



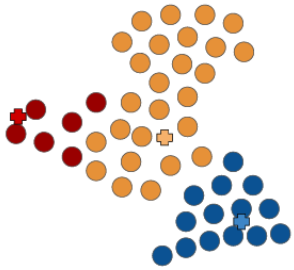
- Hard versus fuzzy
  - fuzzy clustering = object belongs to every cluster, with some weight between 0 and 1 (probabilistic clustering has similar characteristics)
  - in fuzzy clustering, weights must sum to 1

Partitioning clustering



- Partitions a set of N documents into K disjoint clusters
- Find the partition that optimises a specific criterion
- Typically, a function of within-cluster similarity and between cluster-distance

K-means clustering



- One of the most popular clustering algorithms
- Takes a dataset (e.g. documents represented by vectors) and a pre-chosen number of clusters (k) as input
- Outputs the dataset partitioned into k clusters, each cluster represented by its centre point (centroid, shown in the image by crosses)
- The initial centroids you choose are called seeds

Steps to perform k-means clustering

1. Select k cluster centroids (could select k random data points from dataset)
2. Assign each data point to its nearest centroid
3. Recalculate the centroids (as the average of its assigned points)
4. Repeat steps 1 and 2 until convergence (the centroids stop moving or there are only minor movements)

The good and not so good of k-means clustering

- Good: tends to converge quickly
- Not so good:
  - sensitive to choice of initial seeds (may be local minima)
  - but there are some clever methods to pick initial points
- Complexity: O(N * K), where N is number of documents, K is number of clusters

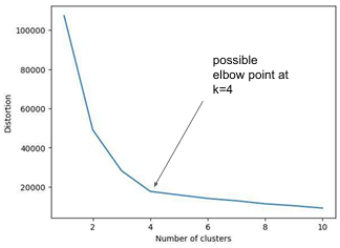
Choosing the initial centroid points for step 0

- Two classic approaches
- Forgy partition:
  - randomly select k existing data points from the dataset and use those as the initial centroids
- Random partition:
  - assign each point randomly to a cluster and calculate the centroids from each cluster

Selecting k is tricky!

- Different values of k can give very different clusters
- Too small k = important clusters are merged, losing information
- Too big k = makes clusters with few data points, too fine grained
- Can visualise and decide k visually for 2D and even 3D data
- We need a general approach for choosing k

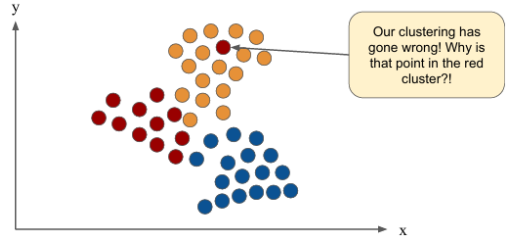
The Elbow method of picking k



- Compute distortion (sum of squared distances of each point to its centroid)
- Plot distortion against number of clusters (k)
- Look for a flattening of the curve (the elbow point) = a good choice of k
  - little gained with a value of k larger than this
- Some arguments against this method

Measuring clustering (how to measure good/bad clustering?)

- Consider a bad clustering:



- Silhouette coefficient for a point i uses:
  - cohesion score (average distance of point i to the objects in the same cluster); we want this to be small
  - separation score (get the average distance of point i to the objects in the other clusters, and take the minimum of these averages)

Calculating silhouette coefficient

- For an individual object i, consider the cohesion score (a) and the separation score (b)
- Silhouette coefficient:
  s(i) = 1 - (a / b) **if a < b**
  s(i) = 0 **if a = b**
  s(i) = (b / a) - 1 **if a > b**
- Silhouette ranges from -1 to +1
- Close to 1 = implies object is in the correct cluster
- Close to -1 = implies object is in the wrong cluster

Web-scale k-means

Algorithm 1 Mini-batch k-Means.

Given: k, mini-batch size b, iterations t, data set X
Initialize each $\mathbf{c} \in C$ with an $\mathbf{x}$ picked randomly from X
$\mathbf{v} \leftarrow 0$
for $i = 1$ to t **do**
  $M \leftarrow b$ examples picked randomly from X
  for $\mathbf{x} \in M$ **do**
    $\mathbf{d}[\mathbf{x}] \leftarrow f(C, \mathbf{x})$ // Cache the center nearest to $\mathbf{x}$
  end for
  for $\mathbf{x} \in M$ **do**
    $\mathbf{c} \leftarrow \mathbf{d}[\mathbf{x}]$ // Get cached center for this $\mathbf{x}$
    $\mathbf{v}[\mathbf{c}] \leftarrow \mathbf{v}[\mathbf{c}] + 1$ // Update per-center counts
    $\eta \leftarrow \frac{1}{\mathbf{v}[\mathbf{c}]}$ // Get per-center learning rate
    $\mathbf{c} \leftarrow (1 - \eta)\mathbf{c} + \eta\mathbf{x}$ // Take gradient step
  end for
end for

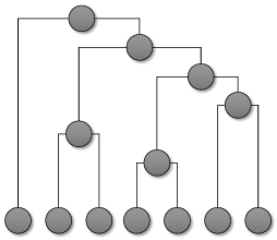
- Running similarity over all items in the dataset is slow, especially for BIG datasets
- We could just compare against a sample
- Mini-batch K-means uses "mini-batches" to reduce computation time
- Mini-batches are random samples of the input data
- Dramatically reduces convergence time, and only slightly worse than running the full algorithm

Summary of k-means clustering

- K-means is the usual choice for an all-purpose clustering algorithm
  - always converges given enough time
  - but could converge to a local minimum
- K-means is fast and efficient
  - could use minibatches
  - search-based retrieval using the centroid as a query
- Possible improvements to the quality of a k-means clustering:
  - pick seeds that are sufficiently far apart (kmeans++)
  - run k-means multiple times with different seeds
  - try different values of k, select the best one
- K-means produces clusters that are:
  - flat (not hierarchical)
  - non-overlapping (disjoint)
  - hard assignments

Extra info on clustering

Hierarchical clustering



- Results in a tree-like representation with clusters of highly similar documents nested within clusters of less similar objects
- Can be agglomerative (bottom-up) or divisive (top-down)

Comparing k-means vs. hierarchical

- K-means is more efficient (O(KN) vs. O(N^3)
- K-means usually produces clusters of a similar quality to hierarchical
- Main drawback = need to define value of k upfront

Cluster evaluation

Look at your data to do cluster evaluation

- Do a smell test (basic checks for oddities, like 90% of data in one cluster)
- Look at a few examples in each cluster (although this is not very systematic)

Intrinsic cluster evaluation

- Check clusters are internally consistent:
  - cohesion (items in a cluster should be similar to each other)
  - separation (clusters should separate objects from each other)
  - silhouette coefficient combines cohesion and separation
- Check clusters are externally consistent (manual human judgements)
  - judge whether pairs of objects are similar enough to be in the same group
  - measure purity (the extent to which clusters contain a single class); this requires human labelled classes

Extrinsic cluster evaluation

- Ask if you are using the clustering for another downstream task (e.g. cluster membership used for a classifier or other system)
- Ask if it improves the performance of the downstream task
- Could compare to random clustering or other methods

Clustering summary

- Clustering is theoretically solid, intuitively plausible
- The above clustering techniques assume each document is only about a single topic (other methods that don't assume this are discussed later)
- Clustering is entirely unsupervised - often more useful to take examples and try to learn predefined groups (more semi-supervised)
- We can cluster any objects not just documents

Week 3 - language modelling [edit]

Probability

Basics

- A probability P(x) must lie between 0 and 1
- The total of all probabilities in an outcome space must total to 1

Joint probability: P(A, B)

- The chance of a set of events happening together
- If the events are independent: **P(A, B) = P(A) * P(B)**

Conditional probability: P(A | B)

- P(A | B) = chance of an event A happening on the condition that B happened
- Bayes theorem: **P(A | B) = (P(B | A) * P(A)) / P(B)**

Probability chain rule

- Can calculate the probability of multiple (not necessarily independent) events all occurring simultaneously
- **P(A, B, C, D)** = P(A | B, C, D) * P(B, C, D)
  = P(A | B, C, D) * P(B | C, D) * P(C, D)
  = P(A | B, C, D) * P(B | C, D) * P(C, D) * P(D)

Independence

- Events A and B are independent if **all three** of the following hold:
  P(A | B) = P(A) // B doesn't influence A **(condition 1)**
  P(B | A) = P(B) // A doesn't influence B **(condition 2)**

  P(A, B) = P(A | B) * P(B) // by chain rule
  = P(A) * P(B) **(this holding is condition 3)**

Probability distributions

- Specifies the probabilities for all possible outcomes of an experiment (or set of experiments)
- Bernoulli/binomial distribution (for experiments with only two outcomes, like coin flips or cards being red, etc.)
- Multinomial (when >2 possible outcomes) (e.g. number on a die, probability of observing a word)

Language modelling

What is language modelling?

- Task of building a predictive model of language
- We are trying to predict the probability of:
  - the next word in a sequence
  - a sentence of words
- Used to be done with language statistics (still useful in many settings, simple and surprisingly effective)
- Recently done with neural networks (very expensive, highly effective)

Language modelling: formal problem definition

- **Important:** language modelling captures both aspects of:
  - syntax/grammar (structure of the language)
  - semantics (meaning / real-world "knowledge")
- A language model is something that specifies the following two quantities for all words in the vocabulary:

  1. Probability of a whole sentence or sequence
  $P(w_1, w_2, ..., w_n)$

  2. Probability of the next word in a sequence
  $P(w_{k+1} \mid w_1, ..., w_k)$

- Note on notation:
  $P(w_1, w_2, ..., w_n)$ is short for $P(W_1 = w_1, W_2 = w_2, ..., W_n = w_n)$
  i.e. the probability of multinomial random variable W_1 taking on value w_1 and so on

How to model language?

- Count (and normalise); need some source text (corpora)
- One way is to estimate the probability of a sequence is the fraction of times you see it:
  $P(w_1, w_2, ..., w_n) = Count(w_1, w_2, ..., w_n) / N$
  where Count = frequency of sequence $w_1 ... w_n$, N is the count of the total number of sequence lengths
- This is the maximum likelihood estimate (MLE) of the (multinomial) probability of drawing this exact sequence from the collection

Issues with direct MLE

- Conditioning on 4 or more words itself is hard because of sparsity
- Estimating probabilities from sparse observations is unreliable
- No solution for a new sequence

Decomposing the probabilities

- Can use the chain rule to decompose joint probability into a product of conditional probabilities
- P("the weather is snowy outside") = P(the)
  * P(weather | the)
  * P(is | the weather)

* P(snowy | the weather is)
* P(outside | the weather is snowy)

- Estimating conditional probabilities with long contexts is difficult
- We can simplify this problem by making assumptions

Markov assumption

- $P(w_{k+1} \mid \mathbf{w_{i-k}, w_{i-k+1}, ..., w_k})$
- Assume next event in a sequence depends only on its immediate past (context)
- Context window in bold
- We only rely on the last "n" words instead of an arbitrarily long context

n-gram models

- A model is often described by its "order" (the size of the context window + 1 / n-gram length)
- Unigram = $P(w_{k+1})$
- Bigram = $P(w_{k+1} \mid w_k)$
- Trigram = $P(w_{k+1} \mid w_{k-1}, w_k)$
- 4-gram = $P(w_{k+1} \mid w_{k-2}, w_{k-1}, w_k)$
- Note: other contexts are possible, and in many cases preferable

Unigram models

- Assumes that next event in a sequence is independent of the past:
  $\mathbf{P(W) = \Pi_i P(w_i)}$
- So in a unigram model, the probability of every word is the product of the probabilities of each word
- An extreme assumption, but can be useful (used in bag-of-words model and similarity)
- Not very good at modelling a sentence, as nonsensical phrases/sentences can get high probability
  - P(the a an the a an the an a) > P(The dog barks)

Bigram models

- Assume next word is dependent on only the previous word
- $\mathbf{P(W) = \Pi_i P(w_i \mid w_{i-1})}$
- This model is fast, efficient, and surprisingly effective

Practical LM issues: start and end tokens

- We introduce special <start> and <end> tokens at the beginning/end of a sequence
- Allows us to model likelihood of beginning/ending of a sentence with a word
- Implicitly the first word for a bigram model is P(w | <start>), sometimes P(w)
- Allows identification of the end of a sequence (e.g. P(<end> | "I like pie.")

Practical LM issues: log space

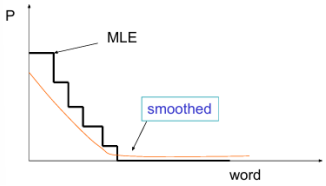
- We do probability computations in log space
  - avoids underflow of very small numbers
  - adding is faster than multiplying
  - sometimes easier to interpret
  - $\log (p_1 \times p_2 \ldots \times p_n) = \log(p_1) + \log(p_2) + \ldots + \log(p_n)$
  - base of log is arbitrary

Smoothing

Smoothing for unseen words

- We need to avoid assigning 0 as the probability of a new word or a new sequence
- We want to assign a low (non-zero) probability to words or n-grams not observed in the text collection (training data)
- Texts are just a sample from the language model
  - missing words should not have zero probability of occuring
  - a non-occurring term is possible (even though it didn't occur in the document or training examples)
- Smoothing = technique for estimating probabilities for missing/unseen words
  - lower (discount) the probability estimates for words that are seen in the document text
  - assign the "left-over" probability to the estimates for the words that are not seen in the text

Discounted smoothing



- Take some probability of the non-zero tokens and give it to the tokens with zero probability

Add-k / Laplace smoothing

- Assume that there were some additional documents in the corpus, where every possible sequence of words was seen exactly k times
- We just increase the frequencies by k to avoid 0 probabilities
- **$P(t \mid \Theta) = (count(t) + k) / (\Sigma_{t'} count(t') + k|V|)$**
- where |V| is the number of unique words in the vocabulary, and the summation is the count of all the other terms added together
- If t is a new/unseen term, we would get:
  $0 + k / (\Sigma_{t'} count(t') + k|V|)$
- We can use add-one smoothing (k=1), but often bad as too much probability is taken from seen events
- We can use a generalised version (adding k instead of 1)
  - note that k could be a partial occurrence (e.g. k=0.5)

Interpolated smoothing

- Back-off or "interpolated" smoothing combines a higher order model with a less sparse model (trigram -> bigram -> unigram) MLE with a model of the collection (training data)
- We are essentially including weighted versions of each probability (e.g. (0.8 * trigram) + (0.15 * bigram) + (0.05 * unigram))
- Example of trigram probability with back-off to bigram and unigram:
  $$P_{smoothed}(w_i \mid w_{i-1}, w_{i-2}) = \lambda_1 P(w_i \mid w_{i-1}, w_{i-2}) + \lambda_2(w_i \mid w_{i-1}) + \lambda_3(w_i)$$
- Subject to all λ_i being >= 0, and they must all add up to 1
- Two other methods that help us pick the lambda values

Jelinek-Mercer smoothing

- Interpolate between the frequency of a token in the document ($P(t \mid \Theta_d)$) and the frequency of a token in the whole corpus (P(t))
- **$P_{smoothed}(t \mid \Theta_d) = (\lambda P(t \mid \Theta_d)) + ((1 - \lambda) P(t))$**

Dirichlet smoothing

- A principled approach to smoothing for a multinomial distribution
- **$P(t \mid \Theta_d) = (count(t, d) + \mu P(t)) / (\Sigma_{t'} count(t', d) + \mu)$**
- Note: the smoothing effectively adds counts to the length of a document
- So, smoothing has greater weight for short documents
- The smoothing parameter μ should be optimised for the task
  - typical default values = {1000 - 2500}

More advanced smoothing

- Many other smoothing models for text (e.g. Kneser-Ney)
- But many have complicated parameters that require tuning
- When in doubt, start simple!
- Stupid backoff used by Google:
  - for computing large n-gram probabilities (up to length 5)
  - if P() over 0 then use it, otherwise use α = 0.4 * next lower-order model
  - e.g. if trigram = 0, use 0.4 * bigram probability
  - keep multiplying by 0.4 (so if you had to drop down twice from a 4-gram, 0.4 * 0.4 * bigram)
  - the end result is a SCORE, not a probability

Evaluation of language models

What is a good language model?

- Should model the language well (i.e. well formed English sentences should be more probable)
- Words that the model predicts as the next in a sequence should "fit":
  - grammatically/syntactically; match the "rules" of the language (P("like i tad") < P("i like tad"))
  - but some syntactically-sound sentences make no sense
  - semantically: conveys actual meaning (P("I saw a supernova at the planetarium") > P("I saw an elephant at the planetarium"))

Evaluating language models

- A trigram model captures more context than a bigram model, so it should be a "better" model, but how do we measure it?
- Extrinsic: measure performance on downstream application
  - e.g. spelling correction, speech recognition, translation, etc.
  - probably most common way to evaluate
- Intrinsic: design a measure inherent to the current task
  - challenging for language modelling

Calculating the probability of "real unseen text"

- We could calculate probability of entire sequence by multiplying all the probabilities of each token together
- But the probability quickly becomes very small
- So we take the per-word probability of a sequence:
  per-word P = P(sequence)$^{1/N}$
- This is taking the geometric mean of the probabilities
- Then we take the reciprocal (1/per-word P) of this to get the perplexity

The equation for perplexity

- **Perplexity =** $1/\sqrt[n]{\prod_{i=1}^{n} P(w_i|w_1, w_2, \ldots, w_{i-1})}$
- where n = length of the sequence
- The token probability from language model could be unigram, bigram, or other
- Low perplexity = language model gave high probabilities for the text, wasn't surprised by it
- High perplexity = language model gave low probabilities for the text, it was surprised by it
- Perplexity ranges from 1 to the size of the vocabulary

Perplexity of a random language model

- In a random language model, the LM gives all tokens a probability of 1/|V|
- Perplexity of random language model = size of the vocabulary |V|

Perplexity of a perfect language model

- In a perfect language model, the LM gives all tokens a probability of 1
- Perplexity of a perfect language model is 1

Practically calculating complexity

- Multiplying probabilities directly = bad idea, because numbers get very tiny and hard to compute
- We work with log-transformed probabilities (multiplication becomes addition)
- Log transformed per-word probability gives an estimate for cross-entropy
  - this has direct links to perplexity

Cross-entropy

- Cross-entropy = measure of the difference between the true probability distribution and the language model's probability distribution
- We treat some "real text" as a sample from the true probability distribution
  - we want a lot of text for a more accurate cross-entropy estimate
- **Cross entropy = -1 * (average log (base 2) probability of each word)**

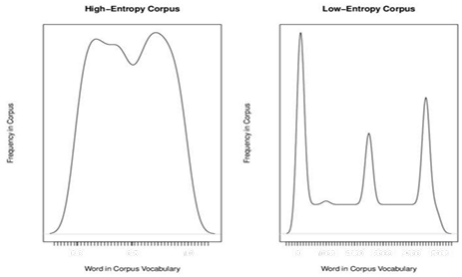
Perplexity from cross-entropy

- **Perplexity =** $2^{\text{cross-entropy}}$
- The above equation uses a power of 2 because we used log base 2 probabilities

Entropy of a probability distribution

- Entropy = measurement of the average uncertainty of information in a probability distribution OR expected number of bits needed to encode a message
- Informally = the total "surprise", adding up the surprise of each possible event weighted by its likelihood
- **H(X) =** $-\sum_x P(x) * log_2(P(x))$

High vs low entropy



- Uniform distribution has high entropy as it is hard to predict a random draw from it
- Peaked distribution has low entropy as it is easy to predict a random draw from it
- Zipf's law tells us: natural language corpora will usually look like the low-entropy graph

Comparing language models with perplexity and cross-entropy

- Perplexity often used to find a "better" language model
  - specific to each dataset
  - summarising a system's ability to model language with a single number is "limiting"
- Cross-entropy relates to the number of bits needed to encode the theoretical model (e.g. 5.00 cross entropy needs 1 more bit to encode than 4.00 cross entropy)

Generating text with a language model

Greedy generation strategy

- We can find the probability of a word in a sequence using a language model:
  $P(w_{k+1} \mid w_1, w_2, ..., w_k)$
- This lets us envision a likely continuation of the text
- We could try and take the word with the maximum probability:

$$\max_{x_1 \in V} P(x_1 | w_1, w_2, ..., w_k)$$ "The dog ___" -> barks

$$\max_{x_2 \in V} P(x_2 | w_1, w_2, ..., w_k, x_1)$$ "The dog barks ___" -> at

$$\max_{x_3 \in V} P(x_3 | w_1, w_2, ..., w_k, x_1, x_2)$$ "The dog barks at ___" -> …

- This is a greedy generation strategy; the actual most likely sequence may be different
  - but it is too expensive to explore the whole space
- This strategy can lead to boring outputs (e.g. the dog that the dog that the)

Sampling from the next token probability distribution

Previous Tokens	P(that)	P(the)	P(dog)	…
the, dog	0.3	0.01	0.02	
dog, that	0.03	0.4	0.02	
that, the	0.02	0.04	0.2	
…				

Previous Tokens	P(that)	P(the)	P(dog)	…
the, dog	0.3	0.01	0.02	
dog, that	0.03	0.4	0.02	
that, the	0.02	0.04	0.2	
…				

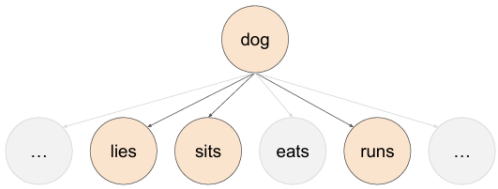
- Instead of picking the highest probability word (left), we can treat the probabilities as a distribution and sample from them (right)
- Means that highest probability token is still the likeliest output, but now the other words have a chance
- Can do some tricks to adjust the probability distribution to give rarer words a better chance
- Or can make choice from only the top k possible words, to stop really outlandish output

One token at a time may not be ideal

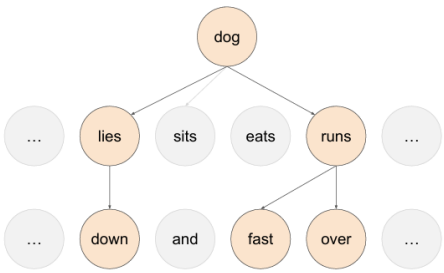
- We are only looking forward for the next word
- But some tokens are rare and unlikely to be generated (e.g. "Barack" coming after "was")
- But a phrase (e.g. "Barack Obama") can be quite common and reasonable to generate
- Very common with names and phrases that start with uncommon words (e.g. esoteric knowledge, serendipitous encounter)
- So we need a way for generation to look a bit further ahead

Beam search

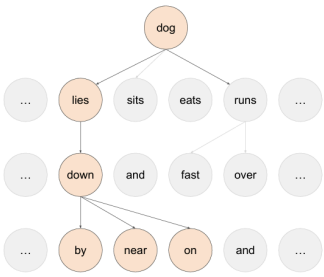
- We can't search all possible generations (even for a small number of tokens) to find the likeliest
- Beam search = grow several candidate short sequences (token-by-token) and discard ones that become too unlikely
- Beam width parameter determines how many tokens to keep



- Here, beam width = 3, so we keep the top 3 tokens and grow those paths



- Add one extra token to the previous beams and keep the paths with the highest probability (here, we only kept the 3 most likely)



- If we extend all the previous paths, keeping only the 3 most likely, we find that the highest probabilities are all sequences starting with "lies down"

More ideas for generating text

- Don't want extremely rare words
  - could filter to only top k words before sampling ("top k")
  - could filter to words with top X% of probability ("top p" / nucleus sampling)
- Don't want it to get repetitive
  - could block repeated n-grams
  - could use advanced contrastive search algorithm (that blocks repeated text that looks similar to previous text)

Language models for document modelling

Modelling documents probabilistically

- Can model a document as probability distribution of tokens
  - unigram distribution of tokens works well, but can use more complex model
- Useful for comparing documents (document similarity)

Unigram language model

- Just uses the probability of each token in a document directly
- Using the count data directly gives us the maximum likelihood estimation for the probability distribution of tokens in a document
- Consider two documents d1, d2:

d1	
END	0.20
the	0.20
a	0.10
frog	0.01
toad	0.01
said	0.03
likes	0.02
that	0.04

d2	
END	0.20
the	0.15
a	0.08
frog	0.01
toad	0.02
said	0.03
likes	0.02
that	0.05

- If we have a string "a toad frog",
  P(string | d1) = 0.10 * 0.01 * 0.01 = 0.00001
  P(string | d2) = 0.08 * 0.01 * 0.02 = 0.000016
- As P(string | d2) > P(string | d1), the first document is more likely to have generated that string
- This principle is useful for a variety of applications, e.g. determining the author of a text when authors of documents are known

- Note: in practical applications log-transforms are used as the probabilities get small

Document smoothing

- Documents are samples of text, won't contain all words in vocabulary
- MLE-based approach would give zero probability for any token not appearing in a document
- Solution:
  - calculate token probabilities across a whole corpus of documents instead of just one
  - or smooth a token probability as a combination of the document probability and the corpus probability
- Consider Jelinek-Mercer smoothing:
  $P_{smoothed}(t \mid \Theta_d) = (\lambda P(t \mid \Theta_d)) + ((1 - \lambda) P(t))$
- Here, the document probability is $P(t \mid \Theta_d)$ and the corpus probability is $P(t)$

Language models and probabilistic similarity

Why use language modelling for document similarity?

- Allows us to incorporate properties of statistical models in a natural way (e.g. smoothing, hypothesis testing)
- "Principled" way of modelling how language is generated
- Instead of looking for how similar two documents are, we look for how likely it is that they both come from the same distribution
- Often more effective than cosine similarity

Kullback-Leibler (KL) divergence

- Gives us a way to define the similarity of two pieces of text in terms of probability
- KL divergence = measure of distance between two probability distributions P and Q
- Information theory: given Q, how many bits (on average) to satisfy P?
- Also called the relative entropy of P with respect to Q

KL divergence for text

- T_p = vocabulary (set) of terms **in document P only**
- $\mathbf{D_{KL}(P \mid\mid Q) = \Sigma_{t \in Tp} P(t \mid \theta_p) * \log_2(P(t \mid \theta_p) / P(t \mid \theta_q))}$
- Range of KL values is non-negative
- KL(P || Q) is smaller if P and Q are more similar
- We use log base 2 because the units are bits of difference
- KL is not symmetric: $D_{KL}(P \mid\mid Q) != D_{KL}(Q \mid\mid P)$

Example: compute D_{KL} (d1 || d2)

- Consider the KL text formula and the Jelinek-Mercer smoothing formula:
  $D_{KL}(P \mid\mid Q) = \Sigma_{t \in Tp} P(t \mid \theta_p) * \log_2(P(t \mid \theta_p) / P(t \mid \theta_q))$
  $P_{smoothed}(t \mid \Theta_d) = (\lambda P(t \mid \Theta_d)) + ((1 - \lambda) P(t))$

- Consider these document vocabularies and the corpus vocabulary:

d1	
END	0.2
frog	0.01
said	0.03
that	0.04

d2	
END	0.2
cat	0.01
dog	0.02
said	0.02
it	0.05
that	0.03

collection	
END	0.2
the	0.2
a	0.1
frog	0.01
toad	0.01
cat	0.02
dog	0.02
said	0.04
likes	0.03
it	0.05
that	0.05

- Consider the string: **frog said that toad likes frog END**
- Remember: **T_p is the set of tokens in document d1 only** here
- Out of the string, only {END, frog, said, that} are in d1
- If we apply the KL text formula only, with no smoothing:
  $D(d1 \mid\mid d2) = 0.2 * \log_2(0.2 / 0.2)$ // END
  $+ 0.01 * \log_2(0.01 \mathbf{/ 0})$ // frog
  $+ 0.03 * \log_2(0.03 / 0.02)$ // said
  $+ 0.04 * \log_2(0.04 / 0.03)$ // that
  = infinity
- We end up dividing by zero, causing the result to be infinity
- We can fix this by smoothing **every probability** we use first
- Let's use $\lambda = 0.8$
- Example for the zero probability P(frog | d2) - **apply Jelinek-Mercer, where Θ_d is d2**
  $P_{smoothed}(frog \mid d2) = (0.8 * (0)) + ((1 - 0.8) * 0.01)$

= 0 + (0.2 * 0.01)
= 0.002

- After doing this for all probabilities and using them in the KL-divergence formula:
  D(d1 || d2) = 0.2 * $\log_2$(0.2 / 0.2) // END
  + 0.01 * $\log_2$(0.01 **/ 0.002)** // frog
  + 0.032 * $\log_2$(0.032 / 0.024) // said
  + 0.042 * $\log_2$(0.042 / 0.034) // that
  ~= 0.049

Jensen-Shannon (JS) divergence

- Symmetric version of KL divergence
- Average KL divergence of P and Q to their mean distribution M
- M = 0.5 * (P + Q)
- The square root of JS divergence is a metric

Week 4 - text classification [edit]

What is classification?

- The task of predicting which of a predefined set of classes (or categories) an object belongs to
- In other words, assigning a class to an object based on some pre-trained knowledge

Classes vs labels

- **Binary classification = each item belongs to exactly 1 out of 2 classes**
- **Multi-class classification = each item belongs to exactly 1 out of >2 classes**
- **Multi-label classification = each item belongs to 0, 1, or many classes**
- 2 possible classes, with objects that have exactly 1 label (e.g. a label true/false) = binary classification
- 2 possible classes, with objects that can have any number of labels (zero up to number of classes) = multi-label classification
- >2 possible classes, with objects that have exactly 1 label = multi-class classification
- >2 possible classes, with objects that can have any number of labels = multi-label classification
- If we have thousands/millions/billions of possible classes, we will need special techniques
  - extreme classification for items with exactly 1 label
  - extreme labelling for items with any number of labels

Supervised learning

Training data

- Classification is a supervised learning task
  - to train a classifier we need a training set with documents for which we know their classes/labels
- Training set is normally labelled by human "assessors"
  - resource intensive task
  - consistency is important with clear classification guidelines

Training data quality vs quantity

- Larger training set = better
- But involves humans = expensive to prepare (unless synthetic data is used)
  - labels must be assigned accurately and consistently for the classifier to learn a true representation of the task
- Much more quantity = training difficulties, need use of specialised hardware (GPUs/TPUs) for hardware acceleration
- Good idea to have multiple assessors label the same items and measure inter-annotator agreement

Training process

- Features from the input documents are extracted
- The features and classes/labels from a training set are fed to a machine learning algorithm to train a classifier, alongside various hyperparameters
- It learns a statistical representation with regards to the salient features of each class/label to be predicted
- After this, your model can:
  - make its own predictions for unobserved (not part of the training set) documents
  - practice against the validation set/dev set to fine tune hyperparameters
- **Evaluation data (e.g. validation set) is constructed the same as the training data, but the classes/labels are NOT given to the model**

Classifying text objects

- We represent text objects (e.g. documents) by their most descriptive features
- Features (aka attributes) can have various types:
  - binary = true/false, 0 or 1
  - nominal (i.e. categories) = UGT, PGT
  - ordinal (categories with ordering) = small, medium, large
  - continuous (i.e. numerical)
- There is no "text" feature type
- We can represent text as:
  - binary features of word occurrences (e.g. one-hot encoding)
  - continuous features (e.g. TF or TF-IDF weights)
  - sequence of tokens
- Each text token is a feature
- Text must be vectorised

Additional features in text classification

- Each classification task requires a different set of descriptive featured that are tailored to the specific task
- Text classification is usually enhanced by adding additional document features
- These additional classification features are hand-crafted to describe the documents (e.g. in spam detection, a binary feature for whether the user emailed the sender previously)
- In author attribution, average word length, average sentence length, and punctuation frequency can be useful additional features

Classification algorithms

Categories of classifiers

- Geometric classification (k-nearest neighbour, support vector machines)
- Probabilistic classification (naïve bayes, logistic regression, neural networks)
- Decision trees are based on both probabilities and geometric interpretation

Majority classifier

- Sometimes called "dummy classifier"
- Assigns the class that appears most frequently in the training data
- Doesn't consider input features, only the labels
- Doesn't really "learn" anything - only useful as a baseline

k-nearest neighbour

- k-NN classifier = "lazy learner", no model is directly built
- Approach:
  - find the k "nearest" samples from the training set (nearest to what we want to label)
  - apply the most common label among these samples
- Another application of document similarity
- Expensive - inference time scales with size of training data

Naïve Bayes (NB)

- Applies language modelling using Bayes Rule
- Estimates the probability of a given token belonging to a specific class/label and the probability of the class itself
- Naïve as it assumes each token is independent
- Advantages:
  - fast and efficient
  - probabilistic
  - interpretable, very simple
  - relatively effective baseline
- Disadvantages:
  - sensitive to errors on rare words
  - weak on rare categories, very large and noisy documents
  - biased due to independence assumptions

Logistic regression (LR)

- A linear classifier, assigns a weight to each input feature for each class, and a weight for each class itself
- To classify, applies a softmax operator over the sum of the scores for each class (produces a probability of each class)
- Regularisation applied to avoid complex weights
- Advantages:
  - very efficient

- works with any feature type (no assumptions about distribution)
- widely used baseline for any classification task

- Disadvantages:
  - typically needs more data than NB to train effectively
  - tends to overfit on data without regularisation
  - regularisation hyperparameters need to be fine-tuned using validation data

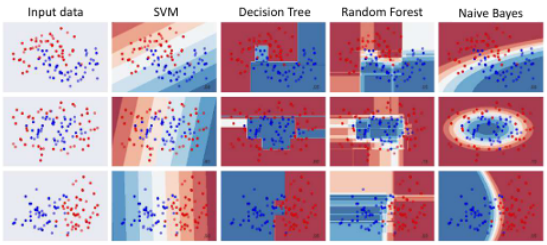
Support vector machine (SVM)

- Identifies the training samples (support vectors) along the decision boundary with the widest possible margin between classes
- Classification conducted by summing the weighted similarity between the target sample and the support vectors
- Advantages:
  - typically more effective than LR for text tasks
  - works with any feature type
  - few hyperparameters to tune
  - fast on small/medium data sets
  - theoretical robustness
- Disadvantages:
  - to handle different types of data, need to select a "kernel function"
  - slow when scaling to large datasets
  - works best with decision boundary is easily separable

Decision tree

- Identifies a set of rules in the form of a tree to make a classification
- Individual trees are often not very robust, so often multiple trees, each with a random subset of features, are ensembled into a random forest
- Advantages:
  - fast, cheap
  - small trees are easy to interpret
  - reasonable accuracy on text tasks
  - can handle a variety of input feature types
- Disadvantages:
  - unstable, small change in data leads to very different tree
  - less accurate than other methods
  - large trees can become complex and difficult to interpret

Which classifier to choose?



- Depends on the case and shape of the data
- Some will be more appropriate than others

How to handle multiclass classification

- Many classifiers work only with binary outputs, we need strategies to combine them
- We can transform the problem to multiple binary classifications:
  - one-vs-all (OvA) = one binary classifier per class; positive is one class, all others are negative
  - all-vs-all (AvA) = discriminate between pairs of classes; (k * k - 1) / 2 classifiers
  - issue: doesn't scale to large numbers of classes (extreme classification)

scikit-learn API for classifying

- Many classifiers within sklearn
- All follow a common API:
  - fit(X, Y) learns a model, where X is size (N_samples * N_features), and Y is an array of size N_samples
  - predict(X) applies the learned model to the (unseen) features in X
  - score(X, Y) measures classification accuracy on the unseen data in X, based on the true labels in Y
  - fit(X) builds a vocabulary on the input data
  - transform(X) creates a document-term matrix (vectorising)

- Be careful with fit_transform(x) - performs both in one step:
  - only run this once to fit on the training data, then use transform
- Classifiers: GaussianNB, LogisticRegression, DecisionTreeClassifier, RandomForestClassifier, SVC (i.e. SVM), SVC(kernel="linear"), SVC(gamma=2), etc.
- Vectorisers: CountVectorizer, HashingVectorizer, TfidfTransformer, TfidfVectorizer

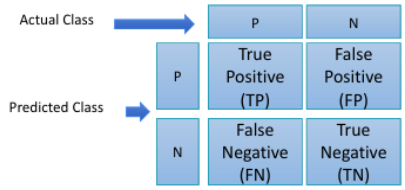
Evaluation of classification effectiveness

Splitting the dataset

- We have a dataset with a set of inputs (e.g. documents) and their assigned classes/labels
- Best practice: split the data into separate training, validation, and test sets
  - training = the classes/labels are used by the learning algorithm to learn a model
  - validation = the classes/labels are used to tune the algorithm
  - testing = classes/labels held out until the very end for evaluation
- Using independent data for evaluation = cross-validation
- Many standard benchmarks already provide splits (good idea to use these to compare your results to others)

Contingency table / confusion matrix

- Evaluate binary classification by constructing a confusion matrix that counts correct/incorrect predictions



- Then select an appropriate evaluation measure for the type of classification task

Classification metrics: accuracy

- Accuracy (Acc) is simple count of correct predictions
- **Acc = #correct / all = (TP + TN) / (TP + FP + FN + TN)**
- Not a great measure (assumes equal cost for both TN and FN)
- Doesn't show class imbalance problem
  - one class may be rare, so most examples of it are negative, few positive examples
  - easy way to get high accuracy in such cases is to always predict the predominant class

Classification metrics: precision and recall

- Precision is "exactness" - what % of elements predicted as positive are actually positive?
- **Precision = TP / (TP + FP)**
- Recall is "completeness" - what % of positive elements are actually labelled as positive?
- **Recall = TP / (TP + FN)**
- Perfect score for both metrics is 1.0
- Inverse relationship between precision and recall

Classification metrics: F-measure

$$F_\beta = (1+\beta^2) * \frac{Precision * Recall}{(\beta^2 * Precision) + Recall} = \frac{(1+\beta^2) * TP}{(1+\beta^2) * TP + \beta^2 * FN + FP}$$

- For binary classification, the standard measure is F_1 (F-measure)
- **F_1** = 2 * (Precision * Recall / (Precision + Recall))
  = 2 * TP / ((2 * TP) + FN + FP)
- F_1 is the harmonic mean of precision and recall
- F-measure can be tuned to give greater weight to precision or recall (lower F-number = more on precision, higher = more on recall)
- F-measure is the most widely used NLP measure, helping to balance both precision and recall

Classification metrics - multi-class and multi-label

- For multi-class, the above accuracy formula is still applicable
  - add up the "correct" classifications along the diagonal of the confusion matrix and divide by the total number of classifications

Classified as	Verb	Noun	Adjective
Verb	19	5	4
Noun	9	25	8
Adjective	16	10	40

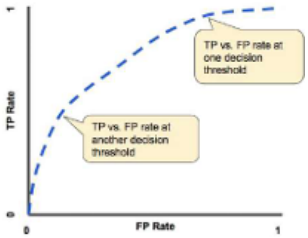
- Effective classifier = large values in the diagonal fields

- Easy to see when classifier is confused about certain classed (hence confusion matrix)
- Multi-label classification is performed as multiple binary classifications
- One confusion matrix for each label
  - not one single confusion matrix for the whole problem (unlike binary and multi-class)
- To evaluate multi-label classification, F_1 is averaged across classes using either micro-averaging or macro-averaging

Multi-class averaging (micro-averaging and macro-averaging)

- Micro-averaging: sum the cells of each type (e.g. TP, TN, FP, FN) across classes before calculating F_1, e.g. $TP = \sum_{c_i \in C} TP_i$ (where c is the set of all cells of that type)
- Macro-averaging: calculate a measure E (e.g. F1) from the confusion matrix of each binary class, and take the average $\sum_{c \in C} E / |C|$
- Micro-average gives equal weight to each instance
- Macro-average gives equal weight to each class

ROC curve and AUC



- Many classifiers produce a score estimate for each class, not a classification
- Can choose a threshold score to pick the class, allowing trade off between precision and recall
- Receiver operating characteristic (ROC) curve = graph showing performance of classification model at all classification thresholds
- Sweep across the threshold and measure TP and FP at each step
- AUC = area under the (ROC) curve
- AUC gives an overview of how well the classifier works across different thresholds

Statistical tests

- Higher F_1 doesn't automatically mean better performing model
- Small differences could be down to random chance
  - e.g. it worked slightly better with the test set, but not representative of it always being better in real life
  - very dependent on size of the test set
- Can use statistical tests to check if an improvement is statistically significant

Classification summary

Type	E.g. Tasks	Classifiers	Metrics
Binary	spam vs not spam, positive vs negative sentiment	All classifiers can be used.	Precision(P), Recall (R), F_1, AUC/ROC
Multi-class	language identification, author attribution (single author)	Naïve Bayes k-NN Decision trees	Accuracy, micro or macro averaged P, R, F_1
Multi-label	image content, author attribution (multiple authors)	Multiple binary classifications	micro or macro averaged P, R, F_1

Machine learning best practices

Train/validation/test splits

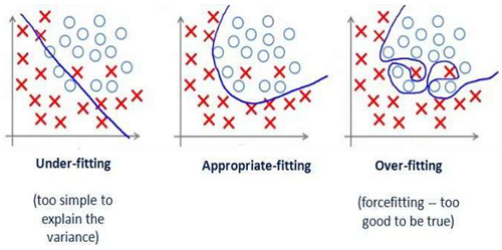
- Test data is sacred
- DO NOT use all data for training
- DO NOT measure (final) performance on training data
- Training = for model construction
- Validation = for setting hyperparameters, error analysis
- Testing = for performance estimation
- Typical starting point for split values: 60%/20%/20% or 80%/10%/10%

Machine learning workflow

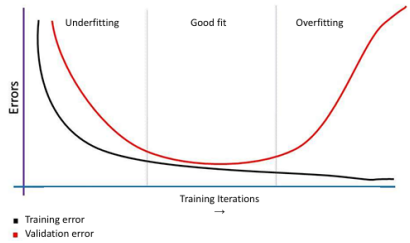
- Create a validation set to tune model (hyper) parameters
  - and develop features

1. Train model on training set
2. Evaluate model on validation set
3. Tweak model according to results on validation set
4. Repeat steps 1, 2, 3 until happy
5. Pick model that does best on validation set
6. Confirm results on test set

Underfitting vs overfitting



- Underfitting = model cannot capture characteristics of problem
- Overfitting = model too closely fits to a limited set of data points



- Can use validation data to prevent overfitting (note validation error begins to increase past certain number of iterations)

K-fold cross-validation

- Technique for when you don't have enough data for full splits
- k=5 or k=10 is most popular
- Randomly partition the data into k mutually-exclusive subsets, each approximately equal size
- At ith iteration, use partition D_i as validation set, others as training set
- Can measure metrics (accuracy/F1/etc) on the validation partitions and take the average
- Still need separate test set

GridSearchCV in scikit-learn

- Used to set parameters of a learner (e.g. svm.SVC()) using cross-validation
- Example: trying lots of C values for a few SVM kernels

```
parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
svc = svm.SVC()
clf = GridSearchCV(svc, parameters, cv=3)
clf.fit(X_train, y_train)
```

Inductive bias in classifying

- Inductive bias = if we don't have data to narrow down the relevant concept to classify for, what are we more likely to prefer?
  - sets of assumptions used to predict the outputs
  - varies by type of learning algorithm

Other problems

- Noise in the training data
  - can occur at the feature level and the label level
  - e.g. label noise: human assessors were not clear on the task
  - e.g. feature noise: typo in sentiment analysis review (e.g. I'm [not] happy ☹)
- Features might not be sufficient for learning
  - e.g. patient's history, genetic data, X-rays, family histories, may still not be enough to predict early stage cancer

Why is my classifier not working?

- Some classifiers need more data than others (e.g. SVM works better on smaller datasets)
- How much data is needed? One line of thinking is much more data than features
- Contrast with common practice: throw in every feature you can think of, let feature selection get rid of useless ones

- Moreover, the presence of irrelevant features hurts generalisation

Feature selection

- Which features are helpful to your model?
- Forward selection:
  initialise s={}
  do:
  add feature to s which improves K(s) most
  while K(s) can be improved
- Backward elimination (or ablation):
  initialise s={1, 2, ..., n}
  do:
  remove feature from s
  which improve K(s) most
  while K(s) can be improved
- Backward elimination tends to find better models
  - better at finding models with interacting features
  - but frequently too expensive to fit the large models at the beginning of search
- Both can be too greedy

Error analysis - what is your classifier good and bad at?

- Essential knowledge for building and deploying an ML system
- Check for obvious biases or other problems
- If still refining model, look for mistakes in training and validation datasets
- If finished and wanting final conclusions, look for mistakes in test set
  - it is bad to use this knowledge to refine your model

Error analysis example

- Consider classifier for business posts that *supposedly* confuses politics with business content
- Get ~100 mislabeled examples from validation set
- Count how many are about politics
- If you only have 5% politics and solve this problem completely, the error will only go down marginally
- If you have 50% politics you can be substantially more optimistic about improving your error

Other tricks for tweaking performance

- Usually very task dependent; look at and understand the data
- Domain-specific features and weights: very important in real performance
  - may need domain expertise to create useful features
  - sometimes need to collapse terms (part numbers, chemical formulas)
  - but stemming doesn't generally doesn't help
- Upweighting: counting a word as if it occurred twice
  - title words, first sentence of each paragraph
  - in sentences that contain title words

Characteristics of well trained ML model

- Fits training set well
- Fits development set well
- Fits test set well
- Performs well in real world

Week 5 - contextual word embeddings: BERT and beyond [edit]

Word vectors (embeddings)

Why do we need word vectors / embeddings?

- Consider TF vectors:
  A = {check: 1, time: 1, watch: 1}
  B = {check: 1, time: 1, clock: 1}
  C = {check: 1, time: 1, elephant: 1}
- Using these for document similarity gives:
  sim(A, B) = sim(A, C) // this is bad

- We need word vectors / embeddings to tell if individual words are similar:
  sim(watch, clock) > sim(watch, elephant)

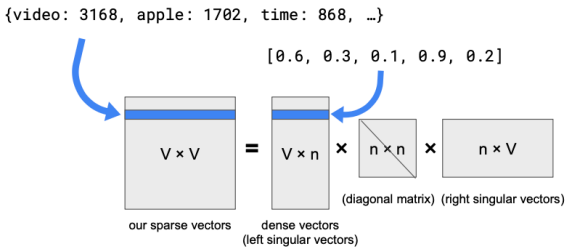
Distributional word vectors

- Imagine a sliding context window across a corpus of documents looking for a word ("watch")

  | context window |
- ...devices watch **video**...
  ...**apple** watch follows...
  ...a watch **time**...
- Sliding the window across all items in the corpus gets us a sparse vector
  watch = {video: 3168, apple: 1702, time: 868, ...}
- We are representing each word as the context it appears in
- We can represent each word as a sparse vector of a context window taken around each occurrence of it in the corpus
- This is called the IBM Model
- These vectors have useful properties (e.g. via cosine similarity)
- Improvements can be made through using TF-IDF etc.

Using singular value decomposition

- Vectors for each term can be large (even when sparse)
  e.g. |watch| = 21,916, where |word| = number of non-zero values
- We can reduce the dimensionality by applying dimensionality reduction techniques like singular value decomposition (SVD):



  (in practice, n is usually in the hundreds of low thousands)
- The reduced vectors (e.g. [0.6, 0.3, 0.1, 0.9, 0.2] are often called static word embeddings
- They maintain the useful properties of the sparse vectors (e.g. sim(watch, clock) > sim(watch, elephant) holds)
- Various other techniques to construct static word embeddings (word2vec, GloVe)

Synonymy and polysemy

- Static word embeddings handle synonymy but NOT polysemy
- Synonymy = when two words have similar/identical meanings
  - similar words will have similar embeddings; dissimilar words will have dissimilar embeddings
  - based on the assumption that synonymous words will appear in similar contexts across a corpus
- Polysemy = when one word has multiple meanings
  - static word embeddings always map the same word to the same embedding
  - a word's embedding can be thought of as the weighted average across all contexts (therefore, less frequent meanings are under-represented)
  - e.g. watch dominated by "view" meaning, under-represents "wristwatch"
  - sim(wristwatch, clock) > sim(watch, clock)
- A word's meaning in isolation isn't as valuable as its meaning in context

Integrating context into vectors

Contextual word vectors (or context vectors for short)

- We turn to deep learning to get vector(word | context)* instead of vector(word)
- Traditional deep learning strategies for working with sequences:
  - convolutional neural networks (CNNs)
  - recurrent neural networks (RNNs)
- Early work (e.g. ELMo) added context to word embeddings using these techniques, but new strategies designed with language modelling in mind brought most of the gains

The big innovations

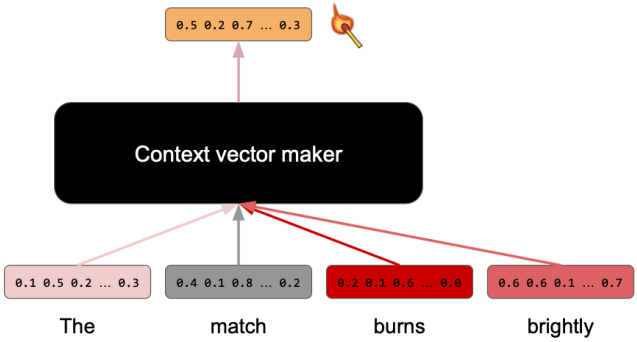
- Self-attention = a neural network structure that builds a new word representation based onto its context
- Subword-tokenisation = limits the size of vocabulary allowing the networks to learn more robust representations
- Transformers = a neural network structure that combines multiple self-attention blocks and allows text encoding/generation

- Language model pre-training = a technique for training neural networks that can be applied to a variety of other tasks
- Prompting = a technique for getting results from a language model without needing to train it for the particular task

Self-attention - what words do you pay attention to?

- Some words around a given word are more important for picking its meaning, e.g. "the match **burns brightly**" (we know its not the sporting event noun or the verb (to be equal to) or the noun for a pair of matching items because of these two words)
- Words just before can tell you about the part of speech (noun/verb/etc) (e.g. "the" implies noun)
- Words across the sentence can identify the sentence topic (e.g. "gas cooker" is very helpful to identify the meaning of "match")
- Many words are filler, not useful for distinguishing meaning

Making a context vector

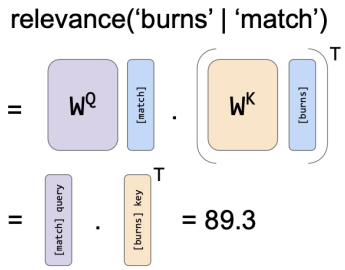


- For a given word, its context vector will be some combination of that word with the other word vectors of the sentence
- We want a function that adds context to word vectors, taking the word vectors without context as input
- e.g. add_context('match' | 'the', 'burns', 'brightly') = F(vector of match | vector of the, vector of burns, vector of brightly)
- But some words are more important than others, so we need relevance weights

Need for a function that tells you how much attention to give

- Calculate the relevance of one word for the context of another word, e.g.:
  relevance('word1' | 'word2') = G(vector of word1, vector of word2) = 12.1
- Takes the word vectors without context as input
- How to calculate relevance from word vectors?
  - can't use similarity (e.g. "match" and "burns" are not very similar)
  - need to do transformations to the word vectors

Relevance for attention



- How do we get the relevance of word1 to understanding word2?
- Input are the vectors for word1 and word2 respectively
- Use two matrices W^Q and W^K (learnt during training)
- Matrix multiply the input word vectors to get a query vector and a key vector
- Dot product these to get the relevance

Relevance scores need to add up to 1

- Relevance scores may not be nicely between 0 and 1
- We want them all to add up to 1, so we use weighting
- We use the commonly used softmax function

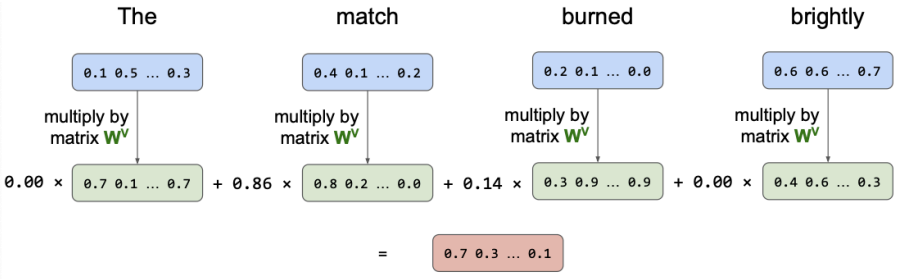
Softmax relevance score

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K.$$

- Means to divide the exponential raised to each element (relevance score) by the sum of all exponentials raised to each relevance score
- Example:
  softmax([12.1, 91.1, 89.3, 44.7]) =
  [($e^{12.1}$ / ($e^{12.1}$ + $e^{91.1}$ + $e^{89.3}$ + $e^{44.7}$)), ($e^{91.1}$ / ($e^{12.1}$ + $e^{91.1}$ + $e^{89.3}$ + $e^{44.7}$)), ($e^{89.3}$ / ($e^{12.1}$ + $e^{91.1}$ + $e^{89.3}$ + $e^{44.7}$)), ($e^{44.7}$ / ($e^{12.1}$ + $e^{91.1}$ + $e^{89.3}$ + $e^{44.7}$))]

  -> softmax([12.1, 91.1, 89.3, 44.7]) = [0.00, 0.86, 0.14, 0.00]

Using relevance scores to weigh the transformed word vectors



- To get the transformed word vectors, multiply them by a matrix W^V (also learned during training) to get their value vectors
- Then add them up, using the softmaxed relevance scores as weights

The self-attention equation using matrices

- **Attention(Q, K, V) = softmax(QK^T / $\sqrt{d_k}$) V**
- where
  Q = input vectors multiplied by W^Q = queries
  K = input vectors multiplied by W^K = keys
  V = input vectors multiplied by W^V = values
- Work with matrices instead of individual vectors
- This equation encapsulates all the previous steps in one go
  - e.g. QK^T is calculating the relevance scores
- Self attention accounts for one extra step:
  - divide by sqrt(d_k) where d_k is the size of the embeddings
  - helps with back-propagation of the network

Where do the embeddings, weight matrices, etc come from?

- Attention relies on multiple matrices W^Q, W^K, W^V
- Each token also has a word vector to use as input
- They are all learned during the extensive training process

Why is it called self-attention?

- Self-attention is named because it is paying attention to the same text as the text it is working on
  - the queries, keys, and values are all from the same text
- Attention can also be applied between texts (e.g. an English text and a Spanish text)

Subword tokenisation

The problem with new words

- New words can occur for lots of reasons (actually new words, misspellings, not in the training set)
- Language models struggle with new words:
  - they know nothing about them
  - have to treat them as out of vocabulary (OOV)
  - no learned embeddings
  - zero probability problem (can solve with smoothing)

Subwords can help us deal with new words

staycation
deepfake
cryptocurrency
microfinance
onboarding
truthiness
annoyingly

- Split uncommon words into 2 or more parts (potentially syllables)

- Depends on language and type of text (e.g. tweets vs science)
- Helps because:
  - much more likely to have seen subwords
  - subwords often give an idea of overall meaning for new words
  - can reduce overall size of the vocabulary (reduces memory needs)

Learning to subword tokenise

- Can use a large corpus of text to learn how to tokenise text into common subwords (e.g. newspapers, tweets, scientific articles, etc)
- Depends on what we want to use the tokenised text for later

Byte pair encoding (BPE)

- Take large corpus of text to learn tokenisation, and a desired vocabulary size as inputs
- Algorithm:
  1. pretokenise corpus documents into words with a tokeniser (removes whitespace)
  2. create a vocabulary of symbols (all unique characters in the corpus, i.e. all letters/numbers etc)
  3. repeat the following steps until the desired vocab size is reached:
     a. find the most common neighbouring symbols (pair) in the corpus
     b. replace all instances of that pair with a new character and add new character to the vocabulary

Byte pair encoding (BPE) example (training)

- Consider the following initial state:

  words in corpus:
  [[p,e,t,e,r], [p,i,p,e,r], [p,i,c,k,e,d], [a], [p,e,c,k], [o,f], [p,i,c,k,l,e,d], [p,e,p,p,e,r,s]]

  current vocabulary:
  [a, c, d, e, f, I, k, l, o, p, r, s, t]

- The most frequent pair in the corpus is ('p', 'e'), with 5 occurrences
- Replace all instances of the pair with 'pe' and add 'pe' to the vocabulary
- New state:

  words in corpus:
  [[**pe**,t,e,r], [p,i,**pe**,r], [p,i,c,k,e,d], [a], [**pe**,c,k], [o,f], [p,i,c,k,l,e,d], [**pe**,p,**pe**,r,s]]

  current vocabulary:
  [a, c, d, e, f, i, k, l, o, p, r, s, t, **pe**]
- **Repeat this with the new most frequent pair until the desired vocab size is reached** (e.g. here it is ('p', 'i') with 3 occurrences)

Byte pair encoding (BPE) example (tokenisation)

- To sub-tokenise a new text, apply the same process (without adding new rules):
  1. pretokenise
  2. split into characters
  3. apply each rule from training (in order)

- Consider initial state:

  vocabulary from training **(learned rules to apply are in bold)**:
  [a, c, d, e, f, i, k, l, o, p, r, s, t, **pe, pi, ck, per, pick**]

  words to tokenise:
  pickle picker

  the words pretokenised and split into characters (subword tokens)
  [[p, i, c, k, l, e], [p, i, c, k, e, r]]

- Applying the first rule (pe) has no effect as it is not in the subword tokens
- Applying the next rule (pi) results in merging of the subword tokens (2 matches):

  [[**pi**, c, k, l, e], [**pi**, c, k, e, r]]
- Do the same for ck, per, and pick -> end result:
  [[**pick**, l, e], [**pick**, e, r]]

Subword tokenisation variants

- Treat the starts of words differently from characters inside the word
  - a '##' prefix indicates a subword inside a word
  - 'pickled' -> 'pick', '##led'
  - 'repickled' -> 'rep', '##ick', '##led'
- Use bytes instead of characters as the initial vocabulary
  - with Unicode, could have many possible characters (e.g. emojis)
  - with bytes, there are only 256 possible values
  - known as byte-level BPE
- Don't pick the most frequent pair to merge
  - pick the pair that maximises the likelihood of the training corpus
  - used by WordPiece method
- Don't do the pre-tokenisation (that removes spaces)
  - include spaces in the vocabulary
  - useful for languages without tokenisers
  - used by SentencePiece method

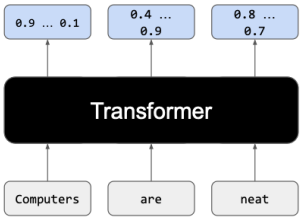
The [CLS], [SEP], [PAD], and [MASK] tokens

- Subword tokenisers for transformers commonly have these 4 special tokens
- [CLS]
  - added at the beginning of sentences
  - used to create a context vector that captures the meaning of the whole sequence
  - helpful for sentence-level CLaSsification tasks
- [SEP]
  - added at the end of a sequence
  - may also be used in between sentences for some tasks
- [PAD]
  - added at the end of a sequence to ensure that multiple sequences are the same length (helpful for batch processing)
  - usually excluded from self-attention computation using an attention mask
- [MASK]
  - used to hide input tokens that need to be predicted (for training a language model)

Training corpus consideration

- Subword tokeniser learned from English newspapers will not work well with German text
- General English subword tokeniser will not work well on scientific text
- Vocabulary and tokeniser depend on the training corpus

Transformers



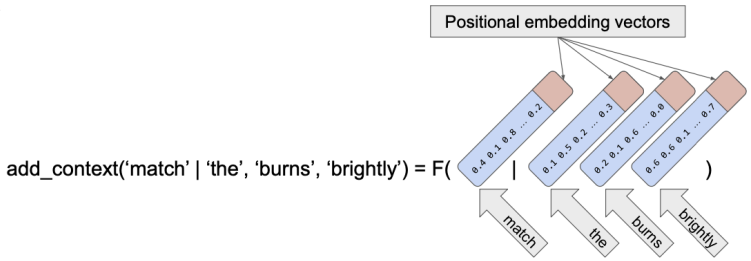
What are transformers?

- A neural network architecture that can encode context, grammar, and other aspects in context vectors
- Good at all kinds of language tasks (e.g. classification, translation, question-answering, etc)
- Combine self-attention, subword tokenisation, and some other neat ideas

Attention (by itself) lacks positional information

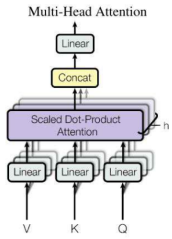
- Location of words relative to each other is important for meaning
  - e.g. if a word is just before another or at the far end of a sentence
- Just adding context doesn't include positional information

Adding positional embedding vectors



- Can encode positions as dense vectors and combine them with existing word vectors
- Different approaches to creating those dense vectors:
  - learning them from scratch
  - using different frequency sinusoidal functions to create vectors that differ at each position in the input

Multi-head attention

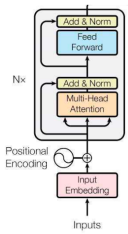


- Attention allows a neural network to focus specifically on certain words for an aspect of language (or so we think)
  - grammar rules, words that indicate a topic, etc
- Single attention system may not be able to deal with all aspects needed
  - e.g. identifying coreference is different to word adjacency
- We can do attention multiple times (many models do it 12 times)
- This is called multi-head attention
  - each attention block has its own W^Q, W^K, and W^V that has different values

Stacking layers

- Deep learning is deep because there are normally many layers
- Outputted context vectors from a transformer layer are fed into the next layer as input successively
- Each layer builds up more meaning (lower layers = basic syntax, higher layers = more complex reasoning)
- Small models have ~12 layers, very big models may have 96 layers

The specifics on a transformer block



- Add = trick for easier training (an optional path that provides a bypass)
- Norm = another trick for easier training (normalises vectors so scaling doesn't go crazy)
- Feed forward network (standard fully connected neural network, allows for encoding more complex functions than attentions' linear combination of inputs)

Training a transformer

1. Provide input and expected output
2. Run input through transformer and compare to expected output
3. Adjust weights in neural network to get output closer to expected output

- Uses backpropagation algorithm
- Inputs and outputs differ based on type of language modelling task

Different language modelling tasks

- Causal language modelling = predict the next word
  - this is a GPT task

- Masked language modelling = predict a masked word (a missing word mid-sentence)
  - this is a BERT task
- Next sentence prediction = predict if one sentence follows another
  - this is a BERT task
- Replaced token detection = spot the corrupted word (a word that shouldn't be there)

Corpora used for training

- Human-created text is used to create examples for language modelling tasks
- Document collections (corpora) getting very big
- May contain multiple languages or even programming languages
- Issue getting "good text" with no problematic language within

Transformers were first proposed for machine translation

- Machine translation = translating one language to another
- Concept of encoder and decoder transformers (e.g. encode English, feed the encoding to a decoder that outputs French)

Encoders and decoders

- Encoder transformers = what we were using to transform input text into context vectors
- Decoder transformers = takes context vectors from encoder, predicts each new token given the previous one starting with <START>
- The output sequence length can be different from the input sequence length

A decoder-only architecture

- Can use decoder transformers on their own (e.g. to predict next word in a sequence)
- Or can use them as part of an encoder-decoder combination

Training the decoder

- We don't want to step one word at a time when training our decoder
- We can do a whole sequence at a time with one trick:
  - the decoder is not allowed to look at future words in the input sequence when predicting a word (as the prediction should only depend on previous words)

Encoder only or decoder only?

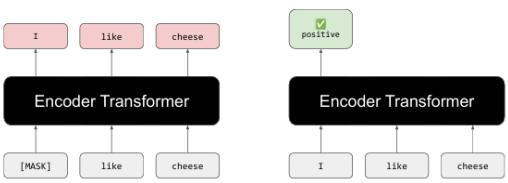
- Encoder architecture only (e.g. BERT)
  - allowed to see all inputs (no masking)
  - good at taking the whole context (previous and future words) into account
  - trained with masked language task
  - called masked language models
- Decoder architecture only (e.g. GPT-3)
  - cannot see "future" inputs, uses clever masking to stop that
  - allows it to get very good at predicting the next word
  - trained with casual language task ("guess the next token")
  - called casual/autoregressive language models

Extra info on transformers

- Bigger models have shown improved performance across many language tasks (e.g. classification, translation, generation, etc)
- Big models require ~600GB of VRAM
  1M parameters = 3.8MB, 1B parameters = 3.7GB
- Transformers have taken over thanks to:
  - brilliant innovations of self-attention, subword tokenisation
  - huge efforts to build massive corpora of text
  - they allow for very big architectures (previous approaches worked serially, one word at a time, but transformers parallelise very well)
  - they can be implemented effectively using GPUs (PyTorch, TensorFlow)

Pretraining and fine-tuning

What is pretraining and fine-tuning?

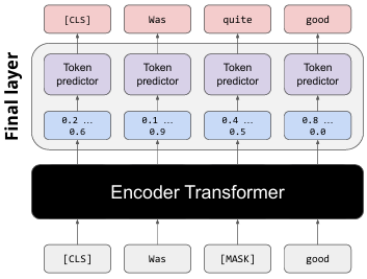


- Pretraining = build a language model that is good at a language modelling task (e.g. masked language model)
- Fine-tuning = re-use most of the deep learning network for another task (e.g. text classification)

Transfer learning

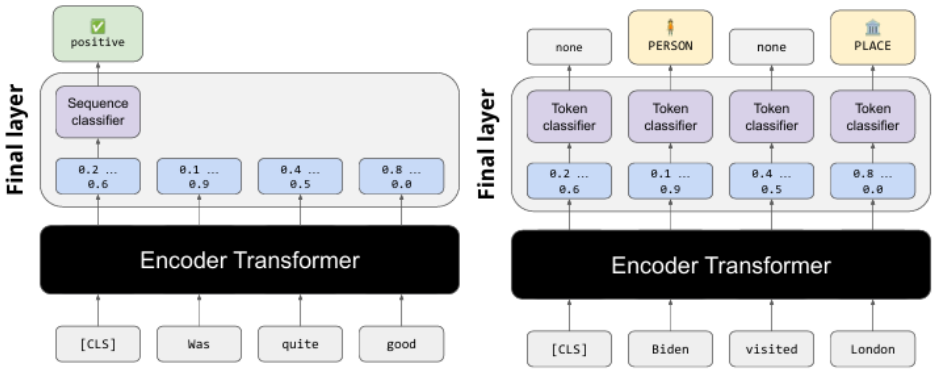
- Train an ML system on one task (e.g. a language modelling problem like masked language modelling)
- Adapt it to another new task (e.g. text classification, summarisation, information extraction, etc)
- Goal is that it performs better on the new task than an ML system that was trained only on the new task
- Pretrained models exist, can download these and finetune them for own needs

Chopping off the head of a transformer



- The head of a transformer is the last layer that makes the final predictions
- In a pretrained model, it is outputting the goal of the language modelling task (e.g. predicting masked words or predicting next token)
- We don't want this output anymore, so we can chop it off and replace it with a new layer for a different task

Swapping in another head



- You can swap in a final layer to fit different tasks (e.g. sequence classifier or token classifier)
- The sequence classifier (a.k.a. text classifier, left) takes only the [CLS] token as input to another small neural network, to predict labels (e.g. positive/negative sentiment if used for sentiment prediction)
- The token classifier determines what entity tokens are (e.g. places/people)
  - this uses the context vector for each token, feeding them into another small neural network that predicts the entity type of the token

Training the new head

- When a new head is added, it doesn't immediately know how to make predictions
- The neutral network for the final predictions is initiated with random weights (so it will initially give random output)
- It needs to be trained
  - you must give inputs and expected outputs
  - you can train only the head (known as freezing the transformer)
  - or you can train whole network (including updating the pretrained transformer)

Week 6 - part-of-speech tagging and parsing [edit]

Background of natural language processing

Word relationships

- Consider this example sentence:
  John drank **some wine** on the new sofa. **It** was red.
- "some wine" is a self-contained phrase (noun phrase)
- The reference of "it" is ambiguous
  - is it a reference (anaphora) to "some wine"?
  - or is it to "the new sofa"?
- Meanwhile, "drank" has direct relationship to its subject "John", and a direct object relationship to "some wine"

- Could consider "on the new sofa" to be a prepositional phrase relative to "drank", that modifies where the drinking was done
- Could say that "on the new sofa. It was red." implies John spilled the wine on the new sofa (pragmatics)
- To understand the relationships in the text, it is important to be able to model them

What is NLP?

- Natural language processing deals with knowledge extraction from text by leveraging syntactic and semantic elements
- Allows:
  - building natural language interfaces for computers/devices in general
  - building intelligent machines which work with (largely textual) knowledge
- Goal is for computers to process or "understand" natural language to perform useful tasks

What does NLP involve?

- Syntax = identifying the grammatical roles of words in sentences (how the words fit together to form valid sentences in a language)
- Semantics = worrying about the meaning of words, phrases, or sentences, and how they relate to one another (who does what to whom?)
- Pragmatics = worrying about how communicative context affects meaning

Context dependence and background knowledge

- One word can change the meaning of another
  e.g. "bank": Rachel ran to the bank vs. Rachel swam to the bank

- John drank some **wine** at the table. It was **red**.
  vs.
  John drank some wine at the **table.** It was **wobbly**.

Garden path sentences

- These are sentences that do not appear to be syntactically correct on the first pass (e.g. "The old man the boat", "The horse raced past the barn fell")

Typical NLP pipeline

- Tokenisation and lemmatisation (harder than you may expect)
- Sentence boundary detection (most NLP operates on individual sentences)
- Part-of-speech tagging (identify nouns, verbs, adjectives, etc)
- Parsing (dependency) - diagramming sentences
- Named entity recognition (detect and classify entities)
- Coreference resolution (resolve pronouns to named entities)

How are the NLP pipeline tasks done?

- Rule-based approaches
  - linguistic + domain knowledge
- Maintaining/updating rules manually is difficult
- Statistical methods for identifying patterns (supervised machine learning)
- Learning from human interactions with computers (mainly as a way to acquire training data)

The supervised ML model approach

- Training:
  1. collect a set of representative training documents
  2. label each piece of text with its label
  3. design feature extractors appropriate to the text and classes
  4. train a classifier to predict the labels from the data
- Testing:
  1. receive a set of test text
  2. run model inference to label each piece of text
  3. appropriately output the model
  4. evaluate correctness of predicted labels

Sequential NLP structures

- Some NLP information (part-of-speech tagging, named entity recognition, word/structure segmentation) can be formulated as a sequence of labels assigned to each unit (token, character, sentence)
- The labels are not independent; they depend on the surrounding labels

Parts of speech (POS)

John	drank	some	wine	on	the	new	sofa	.	It	was	red	.
PROPN	VERB	DET	NOUN	ADP	DET	ADJ	NOUN	PUNCT	PRN	AUX	ADJ	PUNCT

- A very basic syntactic analysis where every word is assigned a tag based on its grammatical role
- Many different granularities, some language-specific
- Generally reasonable to start with "universal" POS tags

Tag	Name	Example(s) - Context-dependent
ADJ	adjective	big, old, green
ADP	adposition (preposition)	in, to, during
ADV	adverb	very, well, exactly
AUX	auxiliary verb	is, has, must
CCONJ	coordinating conjunction	and, or, but
DET	determiner	a, the, some
INTJ	interjection	psst, ouch, hello
NOUN	noun	girl, cat, tree

Tag	Name	Example(s) - Context-dependent
NUM	numeral	five, 70, IV
PART	particle	's, not
PRON	pronoun	he, her, myself, everybody
PROPN	proper noun	Mary, Glasgow, HBO
PUNCT	punctuation	. , ( )
SCONJ	subordinating conjunction	that, if, while
SYM	symbol	$ 😝 ♥‿♥ *1-800-COMPANY*
VERB	verb	drank, running, sits

- State-of-the art techniques achieve >98% accuracy on news
- POS tagging is harder for resource-poor (low training data) languages (~87%)
- POS tagging is harder for tweets (~88% for state of the art)

Named entity recognition (NER)

- Named entities = objects that can be referred to by a name (e.g. "Joe Biden", "MacBook Pro", "University of Glasgow", quantities, dates/times)
- Can do begin-inside-outside (BIO) labelling:

  (B = token starts a named entity, I = token continues a named entity, O = token is not a named entity)

John	Doe	lost	his	£1300	MacBook	Pro	at	the	University	of	Glasgow	.
B	I	O	O	B	B	I	O	O	B	I	I	O

- This is often paired with entity type:

John	Doe	lost	his	£1300	MacBook	Pro	at	the	University	of	Glasgow	.
B-PER	I-PER	O	O	B-MONEY	B-PROD	I-PROD	O	O	B-ORG	I-ORG	I-ORG	O

- Sometimes helpful to treat named entities as a single unit/token (e.g. for text classification, search, etc)
- Entity linking is a related (much harder) task (linking entities to their "canonical" forms)
  - e.g. "Mr. Obama" -> wiki/Barack_Obama
  - depends on context, e.g. football: "Barack" -> wiki/Moises_Barack

Other sequence tagging problems

- Word segmentation

B	B	I	I	B	I	B	I	B	B
而	相	对	于	这	些	品	牌	的	价

- Structure segmentation (e.g. question-answer)

Sequential labelling

Sequential labelling for parts-of-speech, NER, etc

- Build a system that can build these structures (parts-of-speech, NER labels) automatically
  - rule-based: error-prone, hard to build/maintain rules
  - supervised learning: most common/effective approach
- Can learn from training sequences from labelled treebanks

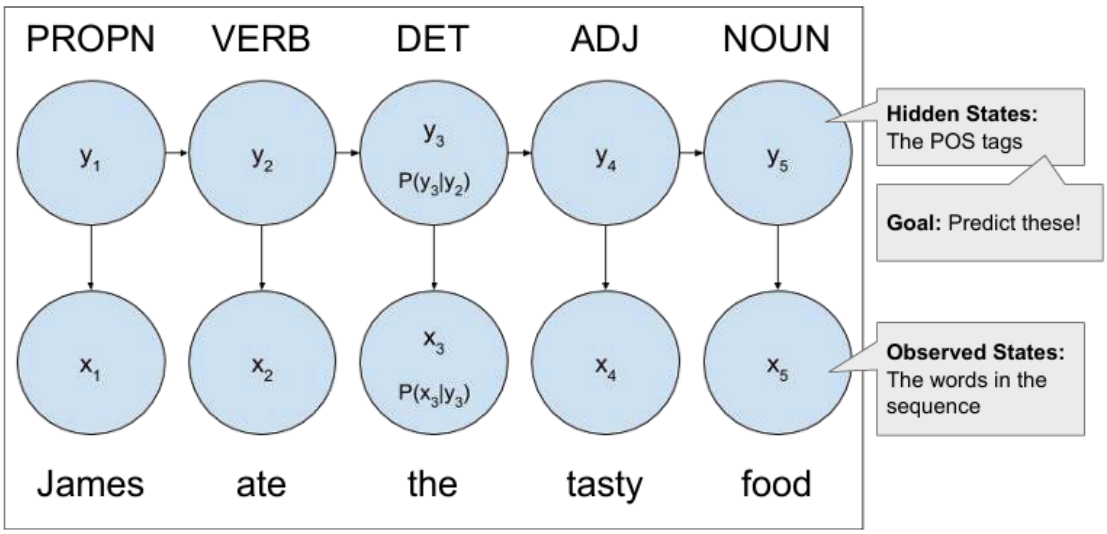
Naïve Bayes for labelling

- Count and normalise probabilities

- $p(C_k \mid \mathbf{x}) \propto p(C_k)\ p(\mathbf{x} \mid C_k)$

  Prob. of POS tag given word | Prob. of POS tag | Prob. of word given POS tag
- But ignores sequential information (same POS tag will be assigned to words regardless of how they were used in the sentence)

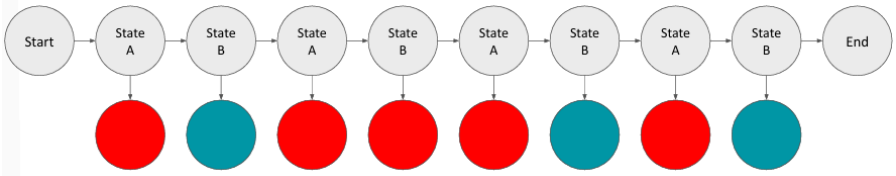
POS tagging with hidden Markov models (HMMs)



- Generalisation of naïve Bayes to sequences
- Assume the y states are hidden (unobserved) states (labels) in which the model can be
- Assume probabilistic transitions between states over time as sequence is generated
- Assume probabilistic generation of tokens from states (e.g. words generated for each POS)
- Assume current state is dependent only on the previous state (the "Markov" assumption)
  - assumption can vary (e.g. 2nd order = previous 2 states)
- Horizontal arrows = **transitions** between hidden states
- Vertical arrows = **emissions** from states (here, each POS)
- x_i = observed nodes (i.e. words)
- y_i = hidden nodes (i.e. POS tags) that we want to predict
- Transitions = $P(y_i \mid y_{i-1})$
- Emissions = $P(x_i \mid y_i)$
- Training: learn $p(y_i \mid y_{i-1})$ and $p(x_i \mid y_i)$
- Inference (test): given x_i, infer y_i or $P(y_i)$

Training for POS tagging with HMMs

- Counting and normalising over the training data gives us the emission and transition probabilities:



P(Red|A) = 4/4
P(Blue|A) = 0/4
P(Red |B) = 1/4
P(Blue|B) = 3/4

Emission conditional probability table		
	Red	Blue
A	1	0
B	1/4	3/4

Transition conditional probability table			
	A	B	<end>
<start>	1	0	0
A	0	1	0
B	3/4	0	1/4

P(A|s) = 1
P(B|s) = 0
P(A|A) = 0/4 = 0
P(B|A) = 4/4 = 1
P(A|B) = 3/4 = 0.75
P(B|B) = 0/4 = 0
P(END| B) = 1/4 = 0.25
P(END| A) = 0/4 = 0

Inference example

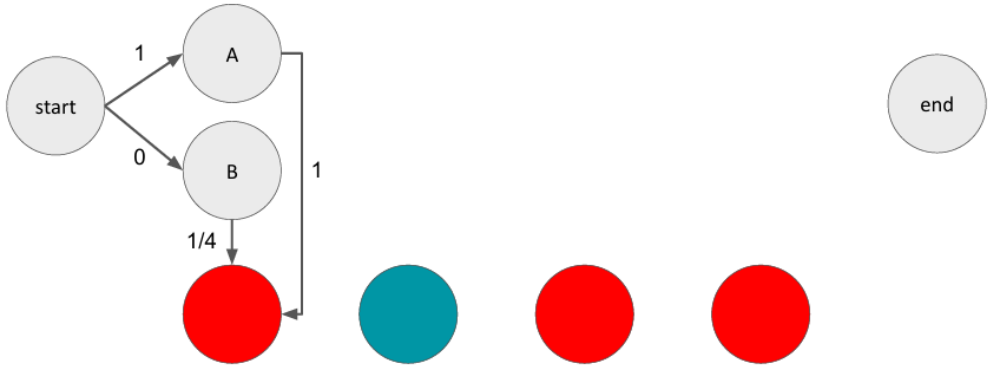
- We are using a greedy approach (making the most likely choice at each outcome without looking ahead)
- Step through the sequence one output at a time and decide the most likely hidden state given the possible paths (transition * emission products)

Emission table		
	Red	Blue
A	1	0
B	1/4	3/4

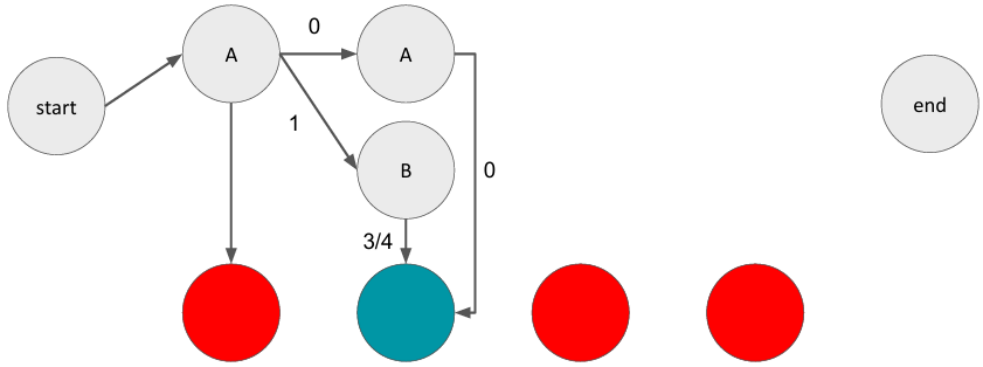
Transition table			
	A	B	<end>
<start>	1	0	0
A	0	1	0
B	3/4	0	1/4

- Transitioning from <start> to A then emitting Red from A
  = P(A | S, RED) = 1 * 1 = 1

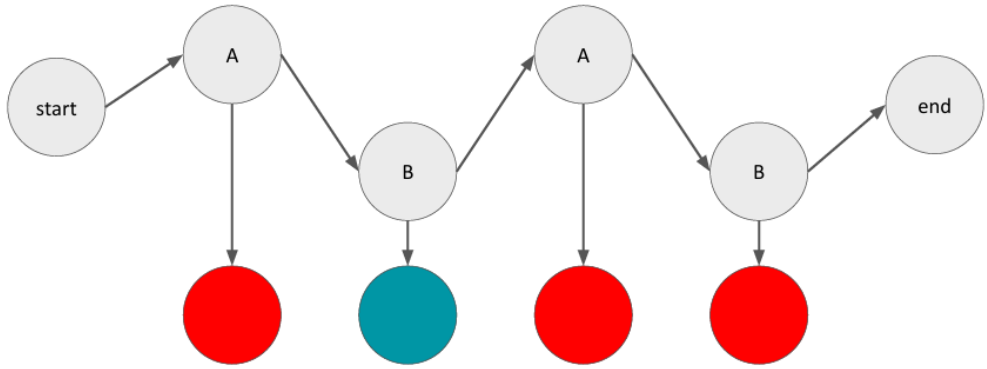
  This is the most likely path (1 > (0 * 1/4)) so we choose it



- Do the same for the next 3 outputs:



- End result:



HMM inference example for POS tagging

- Consider probabilities obtained from training:

Transitions: $P(y_i \mid y_{i-1})$

P(VERB | <START>) = 1/4
P(PROPN | <START>) = 3/4

P(VERB | VERB) = 0
P(PROPN | VERB) = 1

P(VERB | PROPN) = 2/4
P(PROPN | PROPN) = 2/4

Emissions: $P(x_i \mid y_i)$

P(sue | PROPN) = 1/6
P(drew | PROPN) = 2/6
P(mark | PROPN) = 3/6

P(sue | VERB) = 2/6
P(drew | VERB) = 3/6
P(mark | VERB) = 2/6

- Consider sentence "Sue drew Mark"
  - this is really "<START> Sue drew Mark"

- Here are the probabilities (possible paths):

  P(VERB | <START>) * P(sue | VERB) = 1/4 * 2/6 = 2/24
  P(PROPN | <START>) * P(sue | PROPN) = 3/4 * 1/6 = 3/24

  P(VERB | PROPN) * P(drew | VERB) = 2/4 * 3/6 = 6/24
  P(PROPN | PROPN) * P(drew | PROPN) = 2/4 * 2/6 = 4/24

  P(VERB | VERB) * P(mark | VERB) = 0 * 2/6 = 0
  P(PROPN | VERB) * P(mark | PROPN) = 1 * 3/6 = 3/6

  Assignment: PROPN, VERB, PROPN

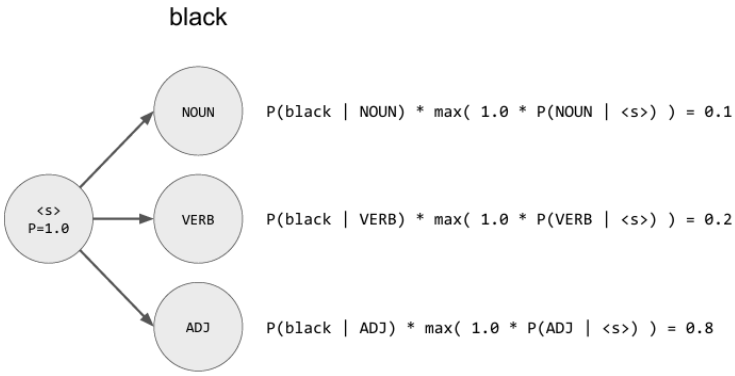
- The most likely tag at each output (set of paths) is used for the assignment
- We go from <START> to a proper noun (PROPN), then check chance of this proper noun being "Sue"
- Then we go from a proper noun to a verb, then check chance of it being "drew"
- Then we go from a verb to a proper noun, then check chance of it being "Mark"

Viterbi algorithm

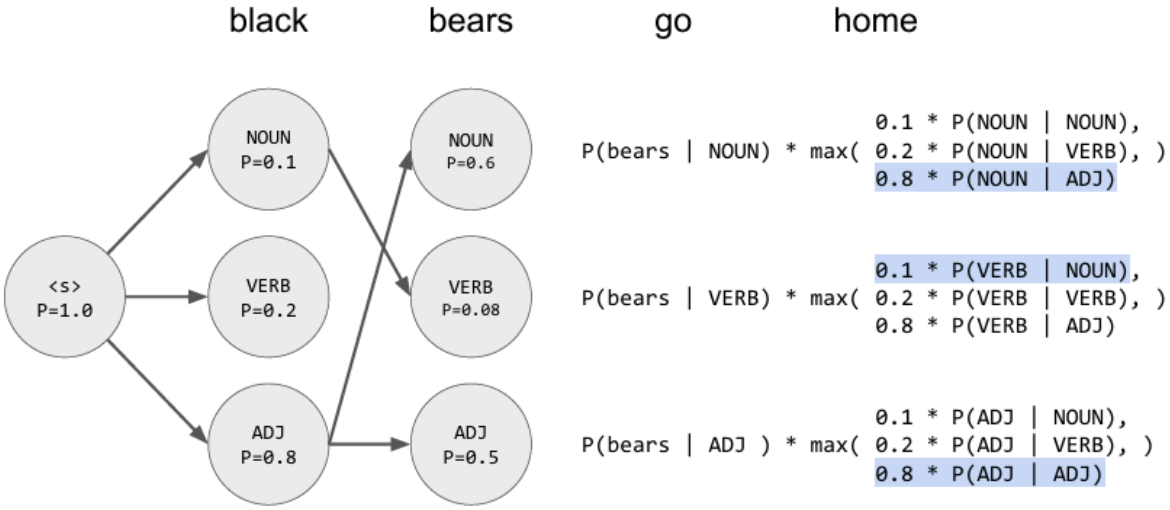
- Unlike the greedy method, takes the whole sequence into account when deciding the most likely tags
- Calculates the maximum possible probability of the entire sequence given transition and emission probabilities
- Builds solution up one token at a time (dynamic programming)
  - then work backwards to find what the optimal path through the hidden states was (giving you the POS tags)
- Algorithm:
  - work through the sequence one token at a time (dynamic programming)
  - use the probabilities of reaching each of the previous states
  - find the most likely transition and emission for this output/emission
  - decide which is the most likely previous state at each step and use that

Viterbi example

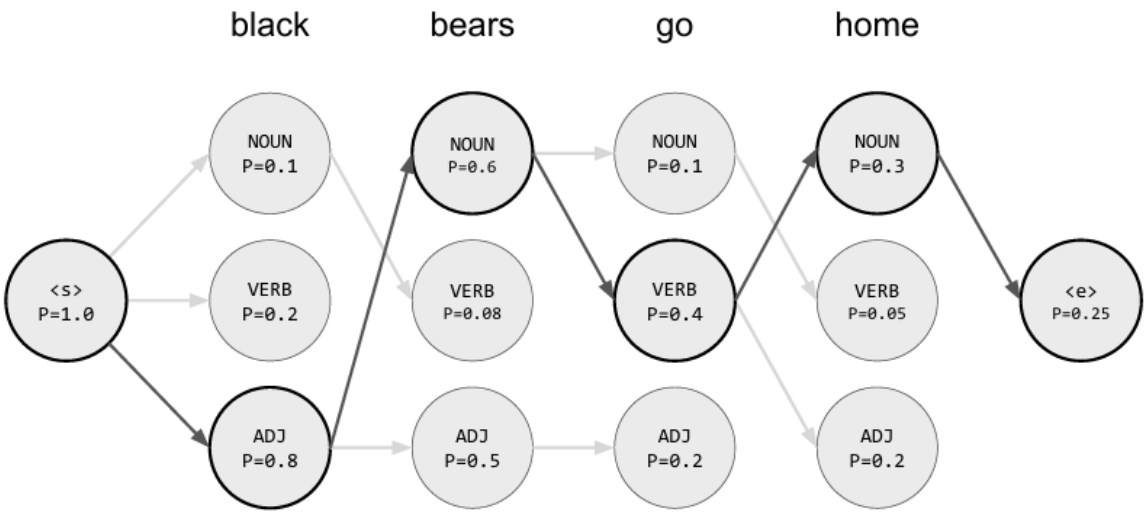
- Consider sentence <s> black bears go home <e>
- We first find the probability of transitions from <s> to each hidden state (of which there can be only 3)



- We keep track of the maximum probability of getting to each hidden state, picking the most likely transitions for each of the 3 hidden states



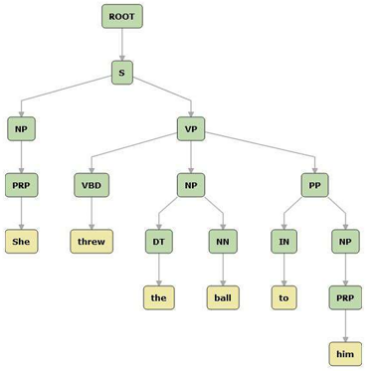
- Simply repeat this for each hidden state for the whole sequence



- Then, if we follow the route backwards from the end, we get the optimal path (and the tags) with the maximum probability 0.25

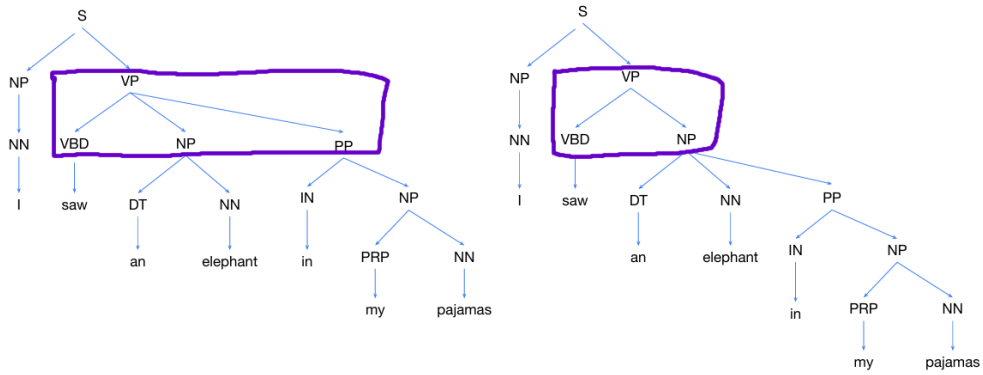
Parsing

Constituency parsing



- Means to break a sentence down into noun phrases, verb phrases, etc
- Noun phrase (NP) = noun with determinants/adjectives ("the small dog", "five cars")
- Verb phrase (VP) = verb + arguments, excluding its subject ("passed David the salt", "walked away")
- Prepositional phrase (PP) = preposition + object (noun phrase)
  - "to the trains", "on top of the Empire State Building"

Ambiguity



- Makes syntactic analysis hard - moderate length sentences (20-30 words) can have hundreds/thousands/tens of thousands of possible syntactic structures

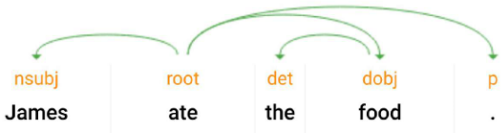
Why do we care about syntactic structure?

- Can learn many aspects of meaning using the syntactic structure
  - NP preceding VP = likely subject of action
  - NP following VP = likely object of action
- Knowing basic units is helpful in modelling language
  - e.g. helpful for predicting/completing sentence (language modelling) or re-organising/simplifying them (summarisation)
- Many NLP problems use sentence structure to make decisions (relation extraction, question answering, machine translation, semantic role labelling, ...)

Parsing: overview

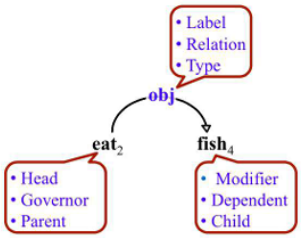
- 3 types of parsing: constituency/phrase-structure, dependency, semantic/frame
- 3 parsing techniques/algorithms: transition-based, graph-based, chart-based
- All try to find a tree that represents how the tokens in a sentence are structured
- All have different formalisms, but use similar algorithms

Dependency parsing



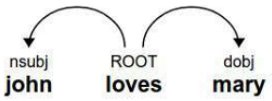
- We draw edges between pairs of words, where the edge labels convey the correct syntactic relation, resulting in a tree
- No single formalism / correct way to draw the parse tree
- Often need training data to train a system

Formal definition of dependency parsing



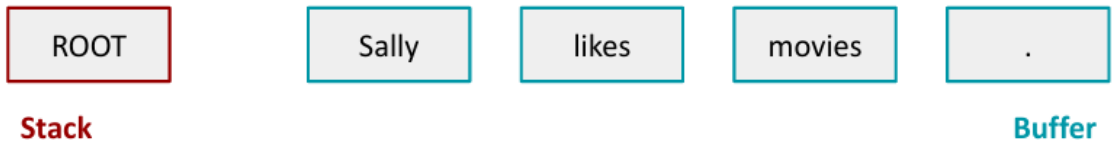
- Dependency structure of sentence = directed graph originating out of a unique, artificially inserted root node (that is always the leftmost word)
- Valid dependency graph conditions
  - each word has exactly 1 incoming edge in the graph (except the root which has 0)
  - weakly connected (replacing its edges with undirected ones results in a connected graph)
  - acyclic

Function labels



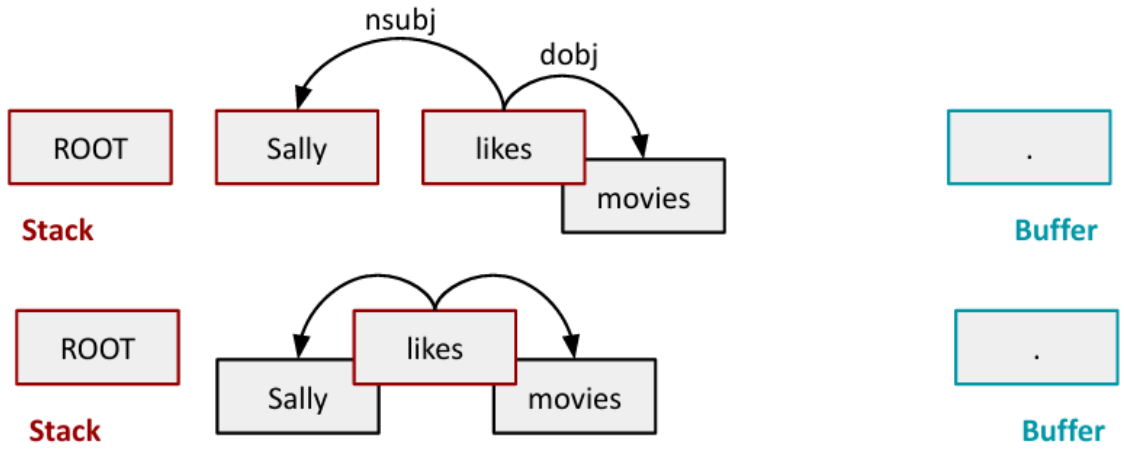
- We can capture linguistic phenomena, like who did what to whom, and label the incoming edges
- Subject = nsubj (noun subject)
- Direct object = dobj
- Indirect object = iobj
- p = punctuation

Transition based parsing

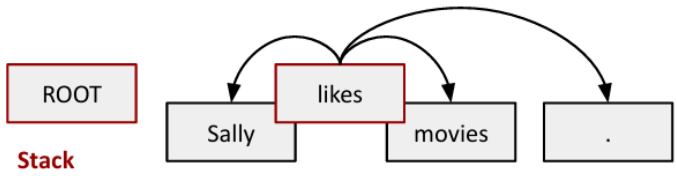
- We build a parse tree with a sequence of actions (transitions)
- Initial set-up state:
  - a buffer initialised with the words
  - a stack to store state (words under consideration for more edges)
  - dependency arcs (edges) = the partially constructed tree
- Arc-Standard parsing - uses these operations:
  - shift (pop from buffer, push to stack)
  - left-arc (add left edge from the topmost word of the stack to the 2nd-top one, pop the 2nd-top one)
  - right-arc (add right edge from 2nd-top word of the stack to the top one, pop the top word)
- 

- Add all words to stack

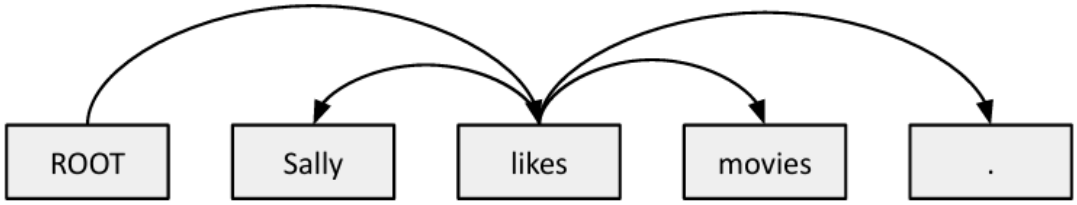
- Do a right-arc then a left-arc



- Add the full stop to the buffer, then do a right-arc



- Then do a right arc from ROOT, which removes likes from the stack, then remove ROOT itself, giving the end parse tree



Transition based parsing: analysis

- The parsing above is also known as "shift-reduce" parsing
  - shift = move from buffer to stack
  - reduce = (left-arc, right-arc), add edges and remove elements from stack
- The parsing is linear (O(N)) as we need to do one shift for each token, one reduce operation for each edge, for N tokens
  - this assumes you can choose the right transitions
  - the actions are irreversible, so you must choose the correct action every time
  - may be more than one correct action (spurious ambiguity)

How to choose transitions?

- First note that this is multi-class prediction:
  - inputs (state = {stack, buffer, other features})
  - targets (transitions = {shift, left-arc, right-arc})
- We can use a classifier, predicting score(a_i | $state_i$) at step i:
  - feed stack features and buffer features to transformer, giving score for each action given current state
  - can use rule-based, SVM, neural network, etc
- Training:
  - predict "gold" actions (from true tree)
  - or "dynamic oracle" rewards good actions

Beyond syntax

- There have been efforts to build semantic structures
- e.g. abstract meaning representations

```
(s / say-01
    :ARG0 (s2 / service
        :mod (e / emergency)
        :location (c / city :wiki "London" :name (n / name :op1 "London")))
    :ARG1 (s3 / send-01
        :ARG1 (p / person :quant 11)
        :ARG2 (h / hospital)))
```

- say-01 = verb disambiguation (PropBank)
  - note the named entity recognition (wiki "London")
- Parsing these is much harder (they are graphs not just trees)

Week 7 - ethics and information extraction

[edit]

Ethical concerns

- may face these
- Models are expensive (GPT-3 = $4.6M, 355 GPU years); prohibitively expensive to train for normal people, unlike companies
- Language model pretraining uses a lot of energy (environmental concerns)
- Input text for training can be biased (e.g. internet text is particularly biased), so output can be biased as the model has encoded bias
- Big data doesn't guarantee diversity (predominantly white and male)
- Question of how to document "unknown" input data
- Chasing model performance leaderboards isn't helpful
- "stochastic parrots" = models are just parrots (they don't understand what they say)
- Are ML systems being deployed too quickly before we understand impact?
- where will the jobs go??

Explainability (interpretation and trust)

The move from features to data

- Models used to be feature-driven
  - relied on human perceived abstract data representations
  - hence more clarity about what happens and how
  - easy to include/exclude features based on intuition
- Now they are data-driven models
  - relies on machine-generated abstractions (e.g. 1D convolution for text, 2D convolution for images)
  - how do we control predictions and convince others of their correctness?

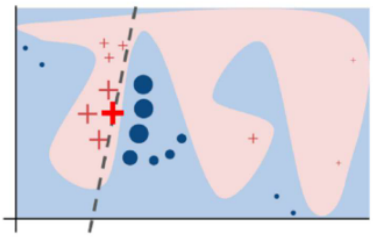
Issues of trust

- If users don't trust a model/prediction they won't use it
- Two notions of trust:
  - trusting a prediction (whether user trusts an individual prediction enough to take action based on it)
  - trusting a model (whether user trusts model to behave reasonably if deployed)
- Both notions impacted by human understand of the model's behaviour, as opposed to seeing it as a black box

Explanation levels

- Prediction-level = explain predictions of any classifier/regression model by approximating it locally with an interpretable model
  - explanations usually in the form of importance weights assigned to different features
- Model-level
  - obtain per-instance explanations or feature weights and then choose a set of representative instances
- Example: tell use the model said they have flu, because we saw {sneeze, headache, no fatigue} (this is LIME explanation)

Local interpretable model-agnostic explanations (LIME)

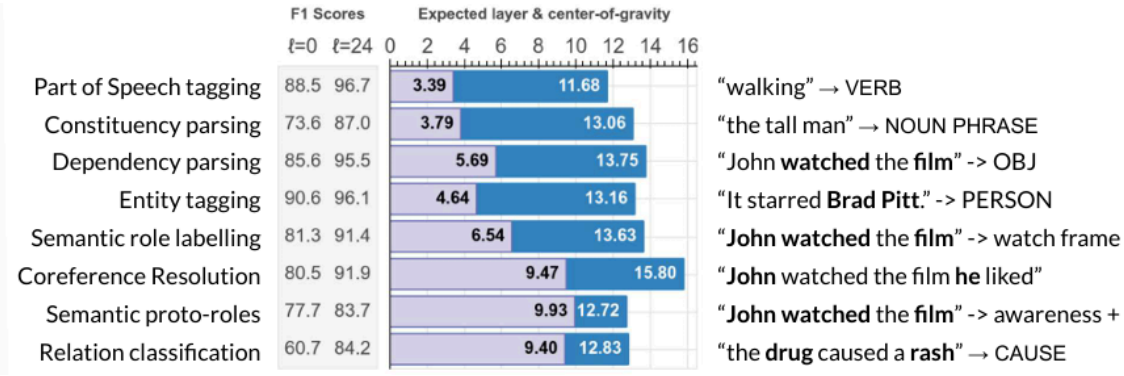


- Estimate importance of each input feature for making a specific prediction (e.g. the red cross)
- Sample points around neighbourhood, fit a simple (linear) classifier on this subset
- The simple classifier approximates the behaviour of the complex boundary locally
- Explain the current point with the parameters of the linear classifier (e.g. a bar chart with 0.31 headache, 0.69 cough)

Shapley additive explanations (SHAP)

- Uses ideas from economics/game theory on the contributions of individuals to an overall task
- Looks at contributions of each feature to final predictions
  - how does the model perform with/without the feature
  - deals with sets of features working together

Understanding how BERT works with explainability



- New language models (BERT/GPT) have impressive capabilities but these are not fully understood (are language models black boxes?)
- BERT learns a traditional NLP pipeline through each of its many layers

Language models for storing knowledge

- Language models seem to encode knowledge, but they can "hallucinate" knowledge by guessing missing words, even if they are factually incorrect
- Important to consider this when comparing with databases

Datasheets for datasets

- ML models trained and evaluated on (usually static) datasets, influenced by training data
- Datasets should be documented (motivation, composition, collection process, preprocessing/cleaning/labelling, uses, distribution, maintenance)
- Ask:
  - who created the dataset and on behalf of who?
  - does it contain all possible instances or just a sample?
  - over what timeframe was the data collected?
  - is the software that was used to preprocess/clean/label the data available?
  - how do you deal with documents of various lengths?

Model cards for model reporting

- When building models, may want to limit use cases
- Model cards are intended to be useful for variety of stakeholder (ML practitioners, software developers, policymakers, impacted individuals, etc)
- Provides details such as:
  - person or organisation developing model
  - model performance measures
  - evaluation data (link to datasheets!)
  - quantitative analyses
  - ethical considerations
  - caveats and recommendations

Information extraction

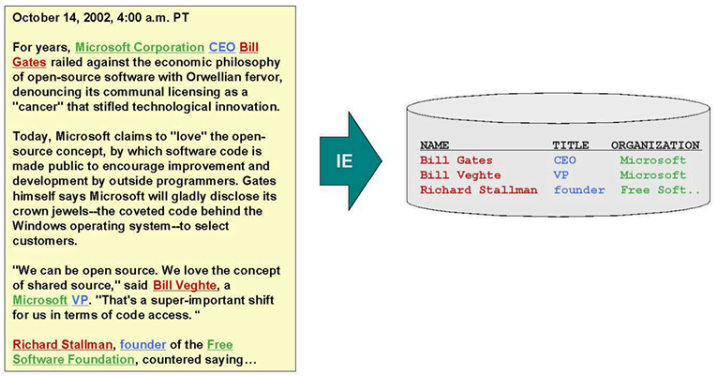
Goals of information extraction

- Organise information so that it is useful to people
- Organise information so that it is useful for machine algorithms (e.g. new knowledge: works-for(x, y) AND located-in(y, z), lives-in(x, z)

What is information extraction (IE)?

- Discover/extract structured information from text by mining lots of information from a corpus ("machine reading" of text)
- To turn unstructured data (documents) into a structured database (aka knowledge graphs)
- Clear, factual information
- Often: <subject><relation><object> RDF triples
  - e.g. <treasure chest><contains_currency><gold>
- Extract entities (e.g. plants, proteins, chemicals, diseases, medicines)
- Extract relations between entities (e.g. contains, is_a, causes)
- Figure out the larger events that are taking place

Information extraction broad example



- Information extraction = segmentation + classification + association + clustering
- Segmentation: first find all the relevant information (e.g. the ones underlined)
- Classification: then classify it (e.g. companies in green, positions in blue, names in red)
- Association: associate together (e.g. {Microsoft Corporation, CEO, Bill Gates})
- Clustering: cluster together (e.g. combine {Microsoft Corporation, CEO, Bill Gates} with {Microsoft, Gates})

Medical information extraction example

- Human summary from textual abstract into structured knowledge summary for machine

Subject	Relation	Object
trap1	is_a	protein
beta-HIVS	is_a	compound
Lithospermi radix	contains	beta-HIVS
beta-HIVS	function_inhibits	v-SRC
beta-HIVS	function_inhibits	EGFR
apoptosis	involved_in	cell_death
beta-HIVS	function_suppresses	TRAP1
ros	function_regulates	TRAP1
…	…	…

Information extraction pipeline

- Named entity recognition - detect and classify entities
- Coreference resolution - resolve pronouns to named entities
- Entity resolution (linking)
  - ground entities to a representation of "world" knowledge
  - e.g. China (LOC) -> the Wikipedia page for China
- Relation extraction: classify relationships between entities

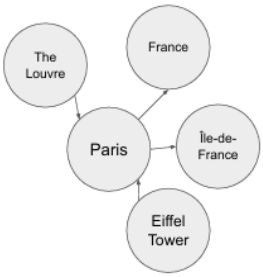
> **President Xi Jinping** of **China**, on his first visit to the **United States** …
>
> <http://en.wikipedia.org/wiki/Xi_Jinping, Office/Leader, http://en.wikipedia.org/wiki/China>

Predefined task vs. open information extraction

- Most IE problems focus on well-defined, predefined tasks (what to include and exclude is decided)
  - e.g. find all mentions of companies in news articles, find all pairs of countries and their capitals from text
- Open IE does not have a predefined task, and tries to turn all text into a structured form

IE to build knowledge graphs / knowledge bases

Knowledge Graph:



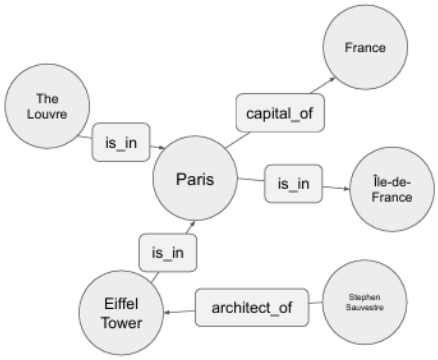
Knowledge Base:

Capital	Country
Paris	France
Madrid	Spain

Museum	City
Louvre	Paris
Kelvingrove	Glasgow

- Using IE to collect enough "fact" triples (two entities and their relationship) = can make a knowledge graph or knowledge base

Knowledge graphs



- Typically directed graphs
- Vertices = entities
- Edges = relations between entities
- Useful for all kinds of applications (search, question answering systems)
- Query = traversal of knowledge graph

Uses of information extraction

- Finding mentions of specific entities (e.g. person, organisation) - can help search functionality
- Find "facts" mentioned in text
- Knowledge bases can help people find information without reading lots of text

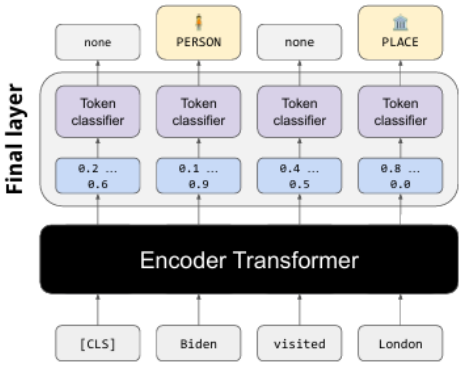
More on named entity recognition (NER)

- NER is a supervised sequence labelling problem (so needs annotated data for training and evaluation)

Conditional random fields (CRF)

- HMMs are the classic NER method but do not perform well with extra features
- HMMs can't (easily) take in extra information
  - part of speech can be useful for predicting NER
  - capitalisation can be a useful signal ("Bath" = city, "bath" = object?)
  - larger context could be useful (what kind of text is this?)
- Basic CRF approach:
  - basic features are current token and previously predicted label
  - **add extra features** (all caps (yes/no), starts with capital (yes/no), part of speech (noun/verb/etc))
  - so turn into supervised classification problem
- CRFs allow modelling of more complex sequences while integrating in extra features

BERT-based sequence labelling



- Can use transformer models like BERT to make token-level predictions
- Also effective: use a CRF on top of the token embeddings

Doing NER with a huge dictionary of terms

- Define a list of terms and synonyms and use exact string matching to find them in the corpus
- Advantages:
  - don't need annotated examples to train a classifier
  - works fairly well for specific cases (e.g. drug names) where words are only used in a single context
- Disadvantages:
  - completely ignores sentence context, requires exhaustive term and synonym list, not appropriate for many purposes

Entity linking

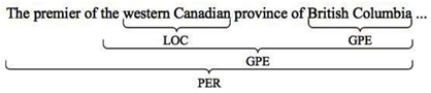
Entity grounding

- Low-level NLP = everything ungrounded (we manipulate text with parsing / part-of-speech, but no idea what it means)
- Information Extraction partly addresses this (strings to things)
- Dealing with entities = can group to "real world" (e.g. Wikipedia entries)

Entity linking formally

- Input:
  a text document D $\{w_1, ..., w_n\}$ of words
  a list of entity mentions $M_d = \{m_1, ..., m_n\}$
- Output:
  list of mention-entity pairs $\{(m_i, e_j)\}$, $i \in [1, n]$
  where each entity is a knowledge base entry (e.g. Wikipedia page) $e \in E$

Difficulties in entity linking



- Nested entity mentions (e.g. 4 nested entities, 3 layers deep)
- Slang entity mentions (Urban Dictionary)

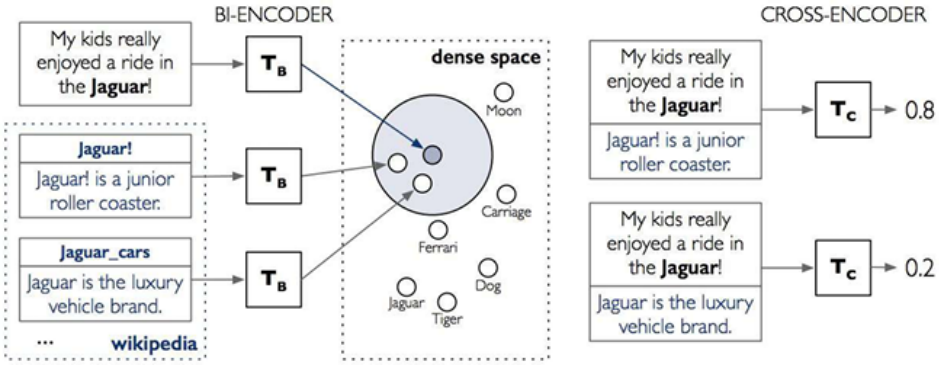
Entity linking as retrieval

- Given a mention, perform search over the KB to find candidates
  - simple string name matching heuristics or retrieval algorithm (BM25, etc)
- Optionally, re-rank candidates
  - possibly "collectively" with evidence from all mentions in D

Using synonyms from the knowledge base

- Many KBs (e.g. WikiData) keep track of the synonyms of each entity
- But what if two KB entities have the same synonym (e.g. Penguin = biscuit or publisher or animal?)
- What if the text matches no synonyms in the KB?

Dense entity representations



- Zero-shot entity linking with dense entity retrieval (using BERT):
  - learn a representation for each entity from text descriptions (cache these)
  - perform approximate "fuzzy" retrieval in dense vector space to find candidates
  - use entity and description representations in final model to score candidates

Week 8 - relation extraction and coreference resolution [edit]

Relation extraction

- Parses text into structured relations to population a database/knowledge base
- Resource Description Framework (RDF) triples
  - <subject> <relation> <object> [optional context]

Located-in(Person, Place)
He was in Tennessee

Subsidiary(Organization, Organization)
XYZ, the parent company of ABC

Related-to(Person, Person)
John's wife Yoko

Founder(Person, Organization)
Steve Jobs, co-founder of Apple...

Why is relation extraction difficult?

- Linguistic variability ("President Obama, "Mr. Obama")
  - address with entity recognition, word embeddings
- Entity ambiguity ("Apple produces seeds", "Apple produces iPhones")
  - address with entity recognition, entity resolution
- Implicit relations ("Obama met with Putin in Moscow -> "Obama travelled to Moscow")
- Complex language with many clauses, long list of qualifiers, negations, etc.

Relations

- Binary relation: relational triple (e_1, relation, e_2)
- Many relations are not binary:
  <company> appointed <person> as <position>
  but can decompose:
  <person><employed-by><company>
  <person><has-job-title><position>
  <company><uses-job-title><position>

Example knowledge bases

- WordNet: low level semantic relations
  - hypernymy (Giraffe, hypernym-of, animal)
  - meronymy (leg, part-of, chair)
- Freebase/Satori/WikiData/Google Knowledge Graph) for world knowledge
- SemMedDB for medical knowledge

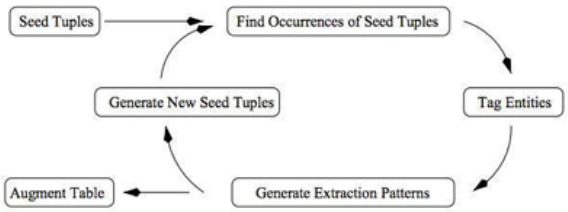
How to extract relations?

"<Barack Obama> is the <president> of the <United States>."

(Barack Obama, head-of-state, United States)

- Find mentions of entities then extract the relations from the context of the entity mentions
- Formulations:
  - rule-based (e.g. Hearst patterns)
  - supervised (extract and score)
  - semi-supervised (bootstrapping)
  - unsupervised (clustering of patterns)

Extracting richer relations using rules

- Multiple relations could exist between entities (e.g. drug (treats/prevents/causes disease)
- How do we know which ones hold?
- Relations often hold between specific entities:
  - located-in(ORGANISATION, LOCATION)
  - founded(PERSON, ORGANISATION)
  - treats(DRUG, DISEASE)
- 

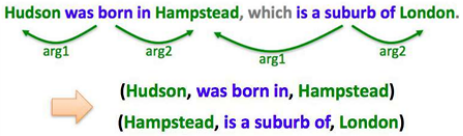
  So we can start by giving input "seed tuples" - relations we already know (perhaps from an existing knowledge graph)
  e.g. located-in(UN, New York City), treats(paracetamol, headache)
- Give the input to generate extraction patterns, for example:
  [ORG] was founded by [PER], patient with [DISEASE] was treated with [DRUG]

ReVerb method for open information extraction

1. Identify "relation phrases" using rules

V	discovered	V = verb | particle | adv
V P	died from	P = prep | partcle | inf. marker
V W* P	played a role in	W = noun | adj | adv | det | pron

2. Extract arguments from relations (also with rules)



Supervised relation extraction

Questions to ask

- What type of supervised task? (sequence classification?)
- What learning model to use? (e.g. BERT, sklearn, consider GPU availability)
- What are your features?

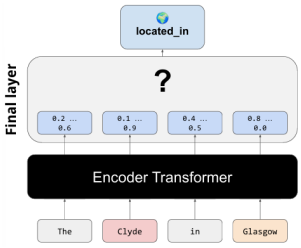
Features example

- **American Airlines** immediately matched the move, spokesman **Tim Wagner** said
- employer(American Airlines, Tim Wagner)
- Could use:
  - bag-of-words features (WM1 = {American, Airlines}, WM2 = {Tim, Wagner})
  - words in between (immediately, matched, the, move, spokesman
  - words before and after (said)

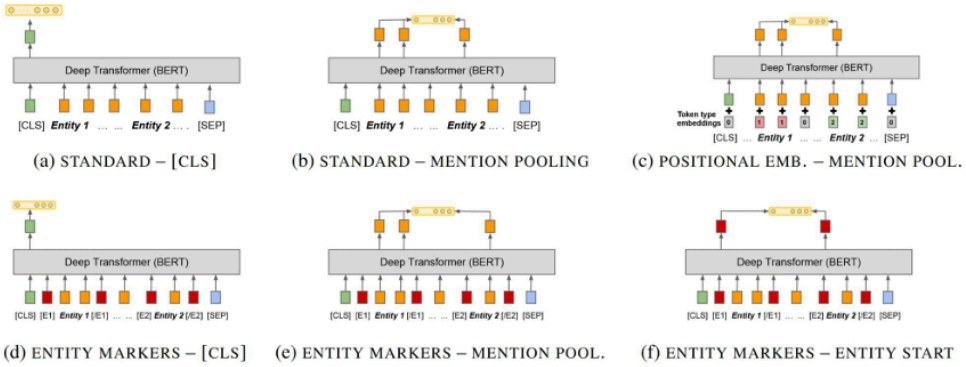
Dependency-parse based

- Not every word in the sentence may be useful for identifying the relationship between two entities
- Can use dependency parsing to isolate important tokens (find the **shortest path** between the two tokens in the dependency graph)

BERT for relation classification



- Need to consider what final layer to use to predict relations (could be multiple possible relations)
- Need to tell the BERT model which tokens are the entities in the candidate relation
- Many ways to represent the relations



(a) STANDARD – [CLS] (b) STANDARD – MENTION POOLING (c) POSITIONAL EMB. – MENTION POOL.

(d) ENTITY MARKERS – [CLS] (e) ENTITY MARKERS – MENTION POOL. (f) ENTITY MARKERS – ENTITY START

- Could just treat as normal text classification without telling BERT about the relevant entities (bad idea, especially if many entities in sentence, as BERT won't know the relevant ones)
- Could use entity markers (special tokens) to identify the entities
- Can use specific context vectors (e.g. the entities or their markers)

Result of relation extraction

- We get large-scale knowledge bases

- Can use them to monitor changes in the world (for fun or profit)
- Can use them to run structured DB queries (data analytics) or answer questions

Distant supervision

- Getting annotated data is costly
- When two known associated entities are in a sentence, they are often discussed with the known relation
- So we could use an existing knowledge base to "label" a lot of text data and use it as training data
- This is "silver standard" data (noisy data)

Coreference resolution

The challenge of coreference

1. [The bee] landed on [the flower] because [it] had pollen.
2. [Bill] was robbed by [John] , so the officer arrested [him].
3. [Jim] was afraid of [Bob] because [he] gets scared around new people

- We don't know which term (purple/green) the pronouns (red) refer to

Why do we need coreference resolution?

- We want to resolve different mentions ("chunk" / noun phrase) of the same entity (a grounded "thing")
- Information extraction, machine translation, question answering systems, chatbots/dialogue systems, other semantic NLP tasks like summarisation, etc)

Coreference process

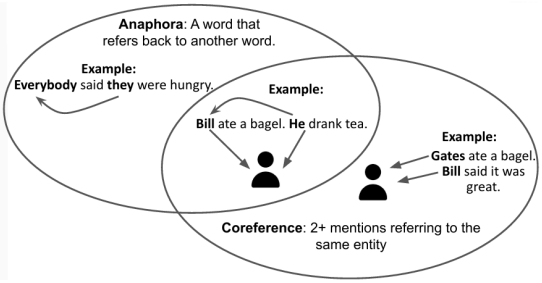
1. Detect the mentions (easy)
   "[I] voted for [Nader] because [he] was most aligned with [[my] values]," [she] said
   - mentions can be nested!
2. Cluster the mentions (hard)
   "[I] voted for [Nader] because [he] was most aligned with [[my] values]," [she] said

Mention detection

- Three types of mentions
  - pronouns (I/you/it/she/him/etc); use a PoS tagger
  - named entities (people/places); use NER tagger
  - noun phrases ("a dog"); use a parser
- Not so simple; should we filter these?
  - it is sunny
  - every student
  - 100 miles
  - the best doughnut in the world

Linguistics terminology for coreference resolution

- Coreference = 2+ mentions referring to the same entity
  - **Barack Obama** travelled to Florida. **Obama** remarked...
- Anaphora: expression depends on antecedent
  - **Sally** arrived, but nobody saw **her**
  (her refers back to Sally)
- Cataphora: expression depends on postcedent
  - Before **her** arrival, nobody saw **Sally**
  (her refers ahead to Sally)
- Exophora: external context (maybe unresolved)
  - He was standing over **there**



Models for coreference

- Rule-based (pronominal / anaphora resolution) models
- Mention pair models
- Clustering models
- Hobbs algorithm:
  - searches parse trees for antecedents of a pronoun, starting at the NP node immediately dominating the pronoun
  - searches previous trees in order of recency, left-to-right, breadth-first
  - looks for the first match of the correct gender and number (male-female/singular-plural)

Coreference resolution problems

- Hard cases (e.g. Winograd schema) require nuanced semantic understanding of the world

Ex: Winograd schema
"The city councilmen refused the demonstrators a permit because they advocated violence."
"The city councilmen refused the demonstrators a permit because they feared violence."

Pairwise coreference model

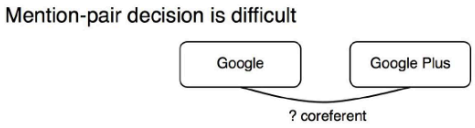
- Given a mention (m_i) and earlier mentions (m_1, ..., m_{i-1}) we classify whether it refers to each entity or not ($m_i == m_j$) given the surrounding context
- This is a binary classification problem

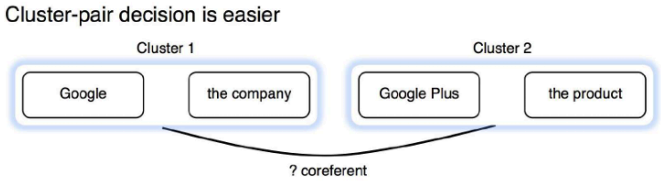
- Mention-pair pairwise coreference models create a coreference chain as a collection of pairwise mention links
- Pairwise coreference models make independent pairwise decisions, and reconcile them in some deterministic way (e.g. transitivity or greedy partitioning); a single mistake can lead to huge (incorrect clusters)

Possible coreference features

- Person/number/gender agreement
  - Jack gave Mary a gift. She was excited.
- Semantic compatibility
  - the mining conglomerate, the company...
- Certain syntactic constraints
  - "John bought him a new car" -> him cannot be John
- We prefer more recently mentioned entities to decide which is referenced
- Grammatical role - we prefer entities in the subject position
- Parallelism:
  - John went with Jack to a movie. Joe went with him to a bar.

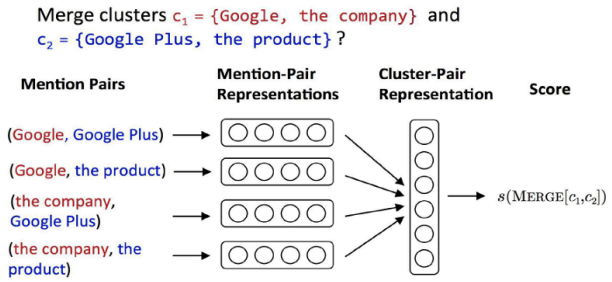
Coreference clustering





- Start with each mention in its own cluster
- Merge a pair of clusters at each step
- Use a model to score which cluster merges are good

Neural model



Evaluation of clustering

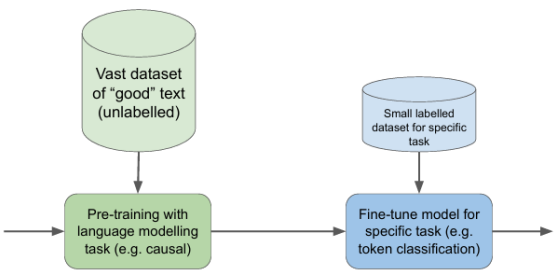
- Hard to compare coreference methods
- Hard to compare correct and incorrect set of clusters
- Many different metrics: MUC, CEAF, LEA, B-CUBED, BLANC
  - we often report the average over a few metrics
- B-cubed - for each mention, compute a precision and a recall
  - precision = % of elements in a hypothesised cluster that are in the true cluster
  - recall = % of elements in a true cluster that are in the hypothesised cluster
  - then average across all the mentions

Week 9 - large language models [edit]

General info on LLMs

- PLM = pretrained language model (BERT/GPT-2/etc), <1B parameters, easily fits on single GPU, used via fine-tuning for specific tasks
- LLM = large language model (GPT-3/4/LLaMA 2/Gemini/etc), >1B parameters, may require multiple GPUs, often used via prompting
- LLMs are all decoder models
- They are trained with the **causal** language modelling tasks (predicting next token)
- They are very good at generating text, much better than masked language models (like BERT) which are trained to guess missing words
  - their context vectors aren't as useful but aren't typically used

Avoiding fine-tuning



- Fine-tuning is expensive computationally and in terms of data
- Really expensive as models get bigger
- Could turn task into a causal language modelling problem, so we can generate text just like the pre-training task
- But it is difficult to turn every task into a causal language modelling problem

- So we can give example instruction inputs and ideal outputs **(training instances)**:

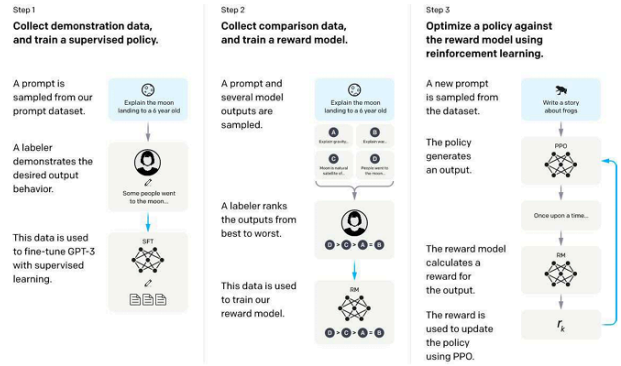
```
Input: The picture appeared on the wall of a Poundland store on Whymark
Avenue [...] How would you rephrase that in a few words?
Output: Graffiti artist Banksy is believed to be behind [...]

Input: I know that the answer to "What team did the Panthers defeat?" is
in "The Panthers finished the regular season [...]". Can you tell me what
it is?
Output: Arizona Cardinals
```

- Lots of datasets of training instances created from existed NLP annotated data with instructions added
- Can generate more synthetic data using LLMs for tuning
  - e.g. use LLMs to paraphrase instructions
- Models can follow instructions beyond the examples
  - it is expected they can follow the same instructions of similar-ish data
  - it is unexpected that they can follow new instructions on different data

Even more info on LLMs

- Instruction tuning is used in all prominent LLMs
  - tune an LLM on many different tasks
  - the instructions from user may be similar to training tasks or the model may be able to extrapolate
- Models can be completion (guess next token of input) or chat (insert input into system prompt template to decide output)
- Reinforcement learning with human feedback:



- LLMs get better linearly with size for some tasks
- Emergent abilities = tasks greater than the sum of their parts (training data) - some argue these are shown by abilities that require a huge size to emerge
- Debate over why very large models are "essential":
  - may emulate more complex reasoning (gaining emergent behaviours)
  - may be able to "remember" enough training data to extrapolate from

Using LLMs

- Zero-shot learning = just explain the task in natural language and give the input data
- Few-shot learning = explain the task as above, but provide examples of the task with expected inputs/outputs, then give the input data
- Difficult to use output from LLMs (paragraphs) in analysis pipelines, may need to constrain generation (e.g. ask for numeric score, then use regex or rules to extract result)

Comparing LLMs to other approaches

- Can compare LLM to supervised approaches with GLUE common benchmark set of NLP tasks
- Fine-tuned BERT beats LLMs (zero-shot ChatGPT) in some tasks
  - few-shot prompting and other techniques can reduce but not close the gap
  - more prompt refinement could help, newer LLMs may beat BERT
- BERT may beat an LLM because:
  - labelled examples are incredibly powerful
  - showing ideal output is the easiest way to describe tasks to a machine
  - few-shot prompting can only use a few examples
  - describing tasks in prompts in detail can be very difficult to achieve (examples are often easier)

Strengths of LLMs

- Work without examples
- Can accomplish very complex tasks that a supervised method would find difficult
- Can work with incredibly diverse inputs (not stalled by common words, often multilingual)
- Will almost always "have a go" and give some (probably okay) output
- Can be used by non-coders

Even more LLM info

- Some language models may be "undertrained" - no need to keep making models bigger
- May be an ideal balance of number of parameters and the amount of text used to train a model to get best performance
- Large language models maybe not trained on enough data, so can get smaller models to compete with big ones with more training data
- Quantisation:
  - allows LLMs to run on CPUs
  - model weights are compressed down (e.g. to 4-bits) instead of the typical 32-bit floating point numbers)
  - output only slightly worse

Prompt engineering

- Get to the point (tell it to give specific output)
- Adding few shot examples can help
- Be precise (many LLMs have large context, so lots of space to specify expected output format and steps to be taken)
- Don't say what not to do in the prompt
- Open research question: should few-shot examples should be similar to the one to be processed?
- Some tasks may not need instructions
- Chain-of-thought prompting = few-shot examples that show "working", seems to benefit models for numerical problems
- Be explicit with the steps for complex tasks and don't expect the steps to be guessed; could break them into a chain of prompts (LLMs solving step-by-step seems to work better)
- Could set up an experiment to compare prompts (need decent amount of labelled data, example inputs/outputs)
- Prompts that work well on one model may not work on another (even for different size-variants of the same model)
- Prompt performance may change with model updates (can freeze models for a few months)

Challenges of LLMs

- Dangerous generations (trained from internet text = bias, some harmful content)
- Can inadvertently or intentionally create harmful content
- Difficult to guarantee harmful content will never be output
- Model alignment = train model to enhance desired behaviours (politeness, clarity, short answers, etc.) and avoid undesired behaviours (e.g. swearing)
  - this generally needs a dataset of human-annotated examples
- Can put safety checks in (separate system that identifies high-risk responses and replaces them with a default response)
  - this also needs a lot of human-labelled examples of high-risk content
- Several harmful content datasets (TruthfulQA, ToxiGen) to test with
- Companies often internally use red teams to test systems (who intentionally try to generate inappropriate content)
- Users can escape safety checks and override alignment + system prompt guidance using specific prompts (jailbreaking)
- Language models can refuse to do legitimate tasks due to safety net rules (rephrasing may help)
- Long conversations can lead to strange output (could just limit chat lengths)
- LLMs have gone off-script if used as chatbots (e.g. for shopping sites)
- LLM performance results can be misleading:
  - ChatGPT trained on internet text
  - likely includes public datasets used for academic research (e.g. sentiment classification)
  - is this cheating if they are trained on the benchmarking data?
- Still expensive + environmental concerns
- Error rate can be hard to measure for complex tasks (e.g. language generation)
- Dependent on large tech companies (who can increase pricing, update/remove models)