Text Preprocessing

Objective:

- We want similar words like 'dog' and 'Dogs' to be considered as the same word to improve the signal in our data and make it less computationally expensive to store every unique word that appears

Tokenization

- splitting raw text into small, indivisible units for processing
- tokens can be words, sentences, n-grams
- tokens can also be characters and patterns found by regular expressions

Parts of Speech Tagging (optional)

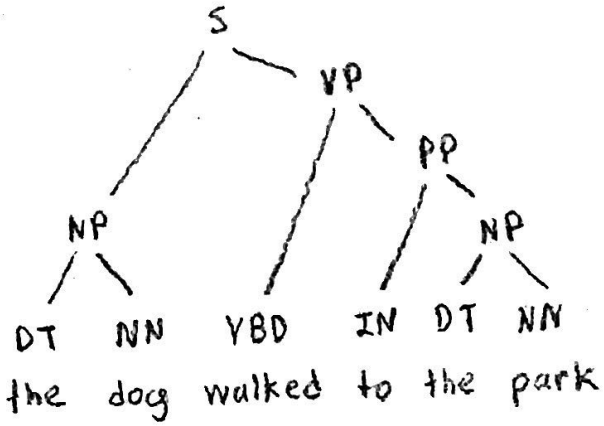
- label each word as a part of speech i.e. nouns, verbs, adjectives, etc
- not necessary to use part of speech tagging, but could help if you are only interested in certain parts of speech like nouns and adjectives to capture context like 'great dog'

Named Entity Recognition (optional)

- identifies and tags named entities in text like people, places, organizations, emails, etc.
- can be used for compound term extraction: 'United States' → 'United_States'

Dependency Tagging (optional)

- identifies relationships between words and modifier words
- useful for tasks like sentiment analysis or aspect mining where surrounding words could offer more context
- can be used for compound term extraction



Removing Characters

- remove punctuation, capitalization, and numbers
- remove stop words, common words that do not add meaning to the text or analysis - It is possible to define your own.

Compound Term Extraction (optional)

- extracting and tagging compound words or phrases in text
- certain words have more conceptual meaning together
  + 'black eyed peas' could be 'black_eyed_peas'

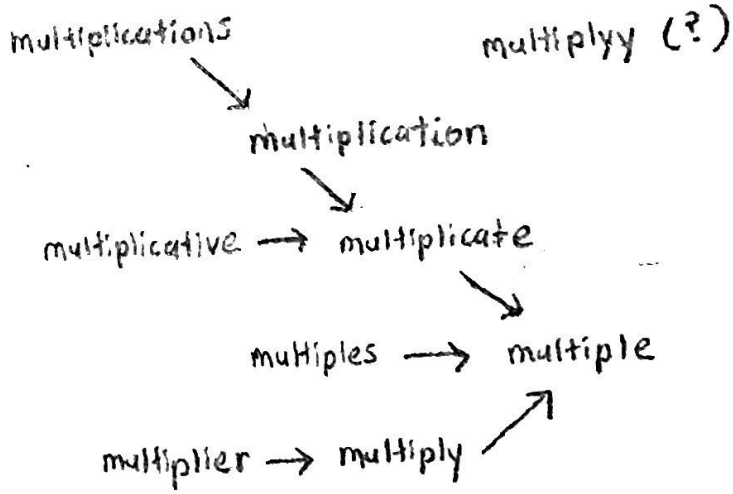
Stemming

- uses rough heuristics to reduce word to base form
- result might not be as interpretible
- could increase recall while harming precision
- better at handling spelling errors

operations	and research	} operat
operating	system	
operative	dentistry	

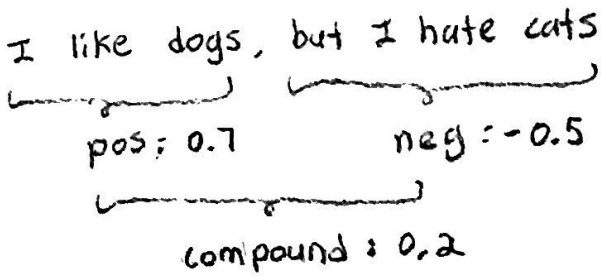
Lemmatization

- uses vocabulary and morphological analysis to reduce word to base form or lemma
- easier to interpret than stemming
- able to recognize words with different tense forms are the same like 'see' and 'saw' both refer to 'see'
- highly dependent on how extensive the dictionary being used is and whether the words are correctly spelled
- could also increase recall while harming precision and it does not do that much better than stemming



Sentiment Analysis (optional)

- not really a text preprocessing step
- extract sentiment about text
- performs better on sentence tokens
- better to use pre-trained package like Vader or Stanford NLP



- 'positive' → positive sentiment, 0 to 1 for Vader with 0 being none
- 'negative' → negative sentiment, 0 to 1 for Vader with 0 being none
- 'compound' → mixed sentiment, with 0 being neutral, <0 being negative, and >0 being positive

Text Vectorization

Count Vectorizer

- the most intuitive approach to representing text as numerical data
- counts the number of times a word occurs for each document in the corpus
- assumption is that word frequency is tied to word importance
- storing extremely low frequency words can be computationally expensive and not helpful in understanding text
  - * better to set min_df, the minimum amount of times a word needs to appear in the corpus to be included in the document-term matrix, for sklearn implementation to an integer greater than 1.
- extremely high frequency words can be uninformative and can worsen model performance by increasing bias e.g. 'and'
  - * normalizing the document-term matrix can help
  - * removing these high frequency words, possibly as stop words, can help if you know which words can be removed
  - * setting max_df, the maximum amount of times a word needs to appear in the corpus to not be included in the document-term matrix, for sklearn implementation to an integer threshold or decimal value less than 1
- which n-gram range to use depends on application
  - * generally, unigrams might not be as helpful, unless some part of speech tagging and filtering is done and you are trying to study adjectives
  - * bigrams and trigrams are more useful for catching common word phrases like 'keep in touch' or obtaining more context about words like 'idea' in the phrase 'great idea'
- Ex.

```
count_vec = CountVectorizer(lowercase = True, ngram_range = (1,3),
                            min_df = 3, max_df = 0.7,
                            stop_words = custom_stop_words)

count_vec.fit_transform(corpus)  # fit only once
count_vec.transform(new_doc)
```

TF-IDF Vectorizer

- gives more weight to words that are rare in the corpus
- consists of two components
  - * Term Frequency : how frequently does a term t appear in the document d

$$TF(t,d) = \frac{\#\text{ of times term } t \text{ appears in document } d}{\text{total number of terms in document } d}$$

  - * Inverse Document Frequency : across all documents in the corpus, how frequently does a term t appear?

$$IDF(t, D) = \ln\left(1 + \frac{\text{total number of documents, } D}{\text{number of documents containing term } t}\right)$$

  - * $TFIDF(t,d,D) = TF(t,d) \times IDF(t,D)$
- parameters and code usage is similar to Count Vectorizer

Word2Vec

- using count/frequency methods for representing text as numerical data can have key disadvantages
  - * text vectors can have high feature dimensions yet be very sparse i.e. many entries in the document-term matrix are 0's
  - * text vectors do not capture semantic relationships i.e. word context from how words are positioned in a document since positional information is not saved
- Word2vec model addresses these key disadvantages
  - * the number of feature dimensions used to define the vector space is pre-set and always fixed, meaning it does not scale with more unique words or a larger corpus
  - * word vectors capture semantic relationships, with vectors that have similar meaning being closer in the vector space - due to how the model is based on word positional probabilities

- the main idea of word2vec is the distributional hypothesis, which states that similar words appear in similar contexts of words around them
- for a word w and a context word w', let us claim that $P(w'|w)$ is the probability of w appearing close to w'
  * for words w' appearing in contexts of man, woman, king, and queen, we might have the following relations:

$$① \frac{P(w'|man)}{P(w'|woman)} \simeq \frac{P(w'|king)}{P(w'|queen)} \quad \text{or} \quad ② \frac{P(w'|man)}{P(w'|king)} \simeq \frac{P(w'|woman)}{P(w'|queen)}$$

  * equation ① states for a topic like gay, the ratio of occurrences of words 'man' and 'woman' is roughly proportional to the ratio of occurrences of 'king' and 'queen'
  * word2vec is trying to perserve word relations through approximate ratios and proportions
  * applying some log transformations to both sides, we get:

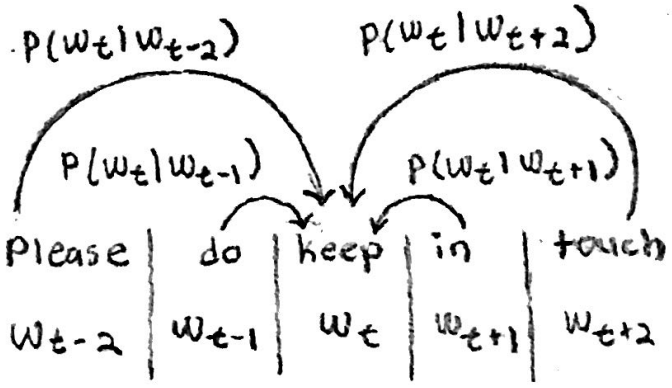
$$\ln\left(\frac{P(w'|man)}{P(w'|woman)}\right) \simeq \ln\left(\frac{P(w'|king)}{P(w'|queen)}\right)$$

$$\ln\left(P(w'|man)\right) - \ln\left(P(w'|woman)\right) \simeq \ln\left(P(w'|king)\right) - \ln\left(P(w'|queen)\right)$$

  * by mapping these logarithms of conditional probabilities to vectors, minimizing the difference between these log terms is equivalent to making these word vectors as close as possible
  * minimizing these differences over a low dimensional space for all word vectors is a gradient descent problem
  * as some word vectors are pulled closer together during optimization, they are pushed away at the same time from other word vectors i.e. pushing in one direction = pulling in another direction when it comes to negative sampling method
  * vectors have magnitude and direction - direction is described by relevancy to feature dimensions i.e. topics while magnitude is associated with word frequency in the corpus

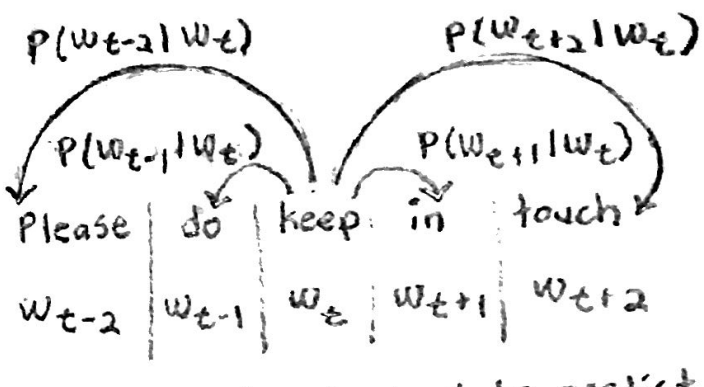
- there are two main types of architecture : CBOW (continuous bag of words) and skip-gram

CBOW



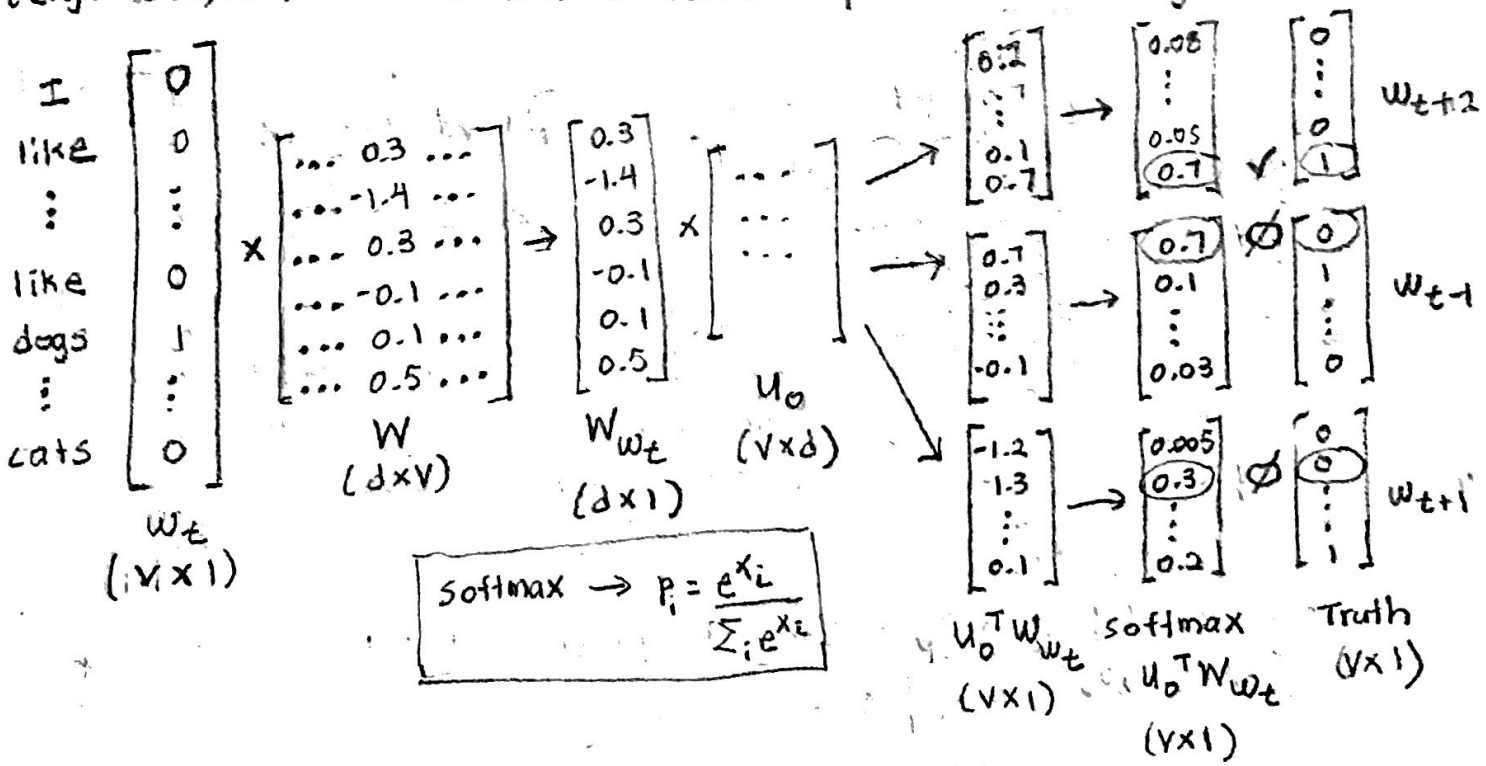
Use surrounding before and after words to predict current center word

Skip-gram



Use current center word to predict surrounding before and after words

- how is skip-gram model trained for V size of vocabulary in corpus (e.g. 500,000) and d size of vector representation (e.g. 300)?



* find 1 hot encoding of current center word
* multiply 1 hot encoding vector by W, the word embedding matrix that contains vector representations of center words, to select W_{w_t}, our vector representation of the current center word
* multiply W_{w_t} by U_o, the word embedding matrix that contains vector representations of context words, to get dot products of center word with each context word, $U_o^T W_{w_t}$

* taking the softmax of the dot products maps the dot products to a probability distribution of 0 to 1, with 1 being the highest probability of a word being a center word
* comparing this to the truth, we can reward the model appropriately
* as it tries to minimize the difference of log probabilities for all the context words i.e $w_{t+2}, w_{t-1}, w_{t+1}, w_{t+2}$
* this becomes a gradient descent problem where the goal is to compute the optimal matrices W and U_o
* traing CBOW is practically the model in the opposite direction

- Skip-gram is typically used for larger corpora, but it is much slower to train since it is essentially multi-classifying
- CBOW is better for smaller corpora
- count/frequency methods might be better than word embedding models like word2vec in certain scenarios:
    * count/frequency methods is easier to implement and determining the right vector representation size for word2vec can be difficult
    * pre-trained word2vec models do not work well if your dataset is small and context is domain-specific
    * word embedding models can be computationally expensive to train, especially for larger corpora
- other variants like doc2vec work well for sentences or documents where similar terms appear in atheist and religious texts, but from context, those texts are different because of how they use those terms
- Glove is similar to Word2Vec, but it is not trying to predict words
    * Word2Vec captures co-occurrence of words one window at a time
    * GloVe captures co-occurrence for the overall corpus
    * GloVe does not use neural nets, the gigantic co-occurrence matrix is just factorized to produce a lower-dimension representation ⟶ still gives word vectors

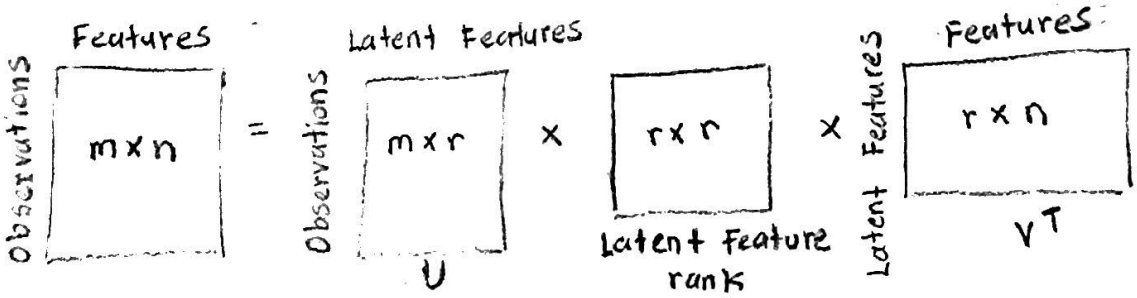
Dimensionality Reduction

Assumptions for Dimensionality Reduction

- assume our data mostly lies in a lower dimensional space i.e can be explained with fewer feature dimensions
- curse of dimensionality : as the number of feature dimensions increases, the data becomes increasingly sparse and the model is unable to explain all the variance in the data
- simply removing feature dimensions might cause us to lose to much info
- dimensionality reduction can help us perserve info while reducing the number of feature dimensions so our data is not so sparse
- the latent features produced through dimensionality reduction could encode important info while removing dimensions with low variance, which are not that helpful for modeling since it offers little data separability

Singular Value Decomposition (SVD)

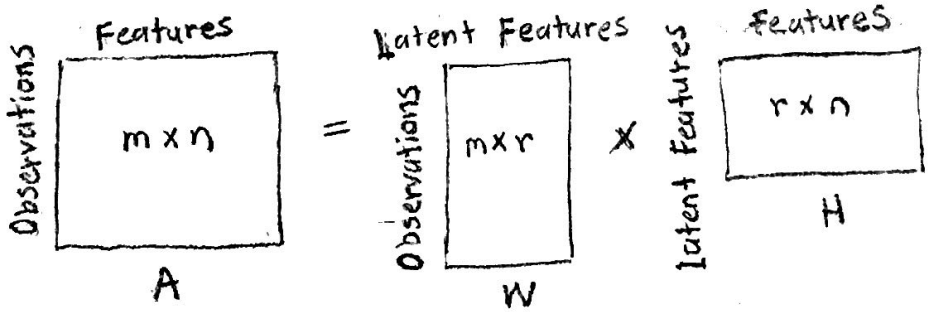
- use matrix decomposition on data to perform feature extraction
  - * $A_{m \times n} = U_{m \times r} \Sigma_{r \times r} V^T_{n \times r}$
  - * A is our data matrix where we have m number of observations and n number of features
  - * U is the left singular vector matrix and it shows how our observations are related to the new latent features
  - * Σ is the diagonal matrix of singular values and it tells us how much variance can be explained by a latent feature with a value of 0 being unhelpful. It is essentially the latent feature weight or importance (off-diagonal elements are 0)
  - * V^T is the right singular vector matrix and it shows how our features are related to the new latent features



- Σ is useful for latent feature selection since latent features are in order from highest to lowest weights across the diagonal, from the top left of the matrix to the bottom right
    * we can further reduce dimensionality by dropping latent features in U with low weights in Σ by setting the weights to 0
- U is the matrix we want in this context for modeling on latent features and observations
    * Note: sklearn implementation of LSA gives you U as output
- how many latent features to keep depends on context, but enough to explain ~70-80% of variance in data

Non-Negative Matrix Factorization (NMF)

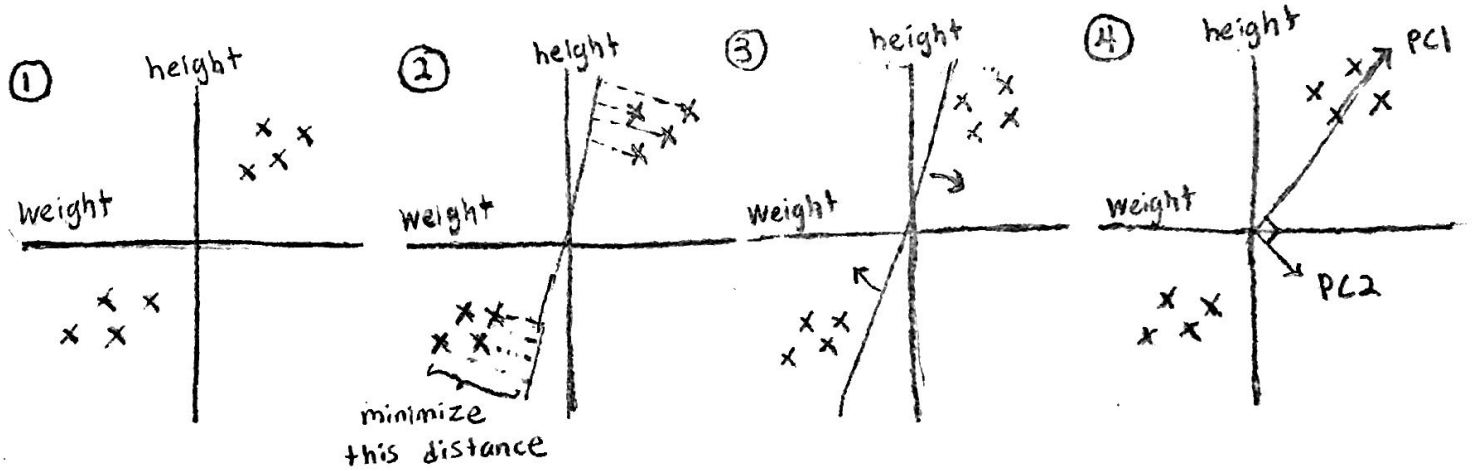
- matrix decomposition that produces 2 matrices with non-negative values
- $A_{m \times n} = W_{m \times r} H_{r \times n}$
    * A is our data matrix where we have m number of observations and n number of features
    * W shows how the observations are related to the latent features
    * H shows how our observations are related to the new latent features



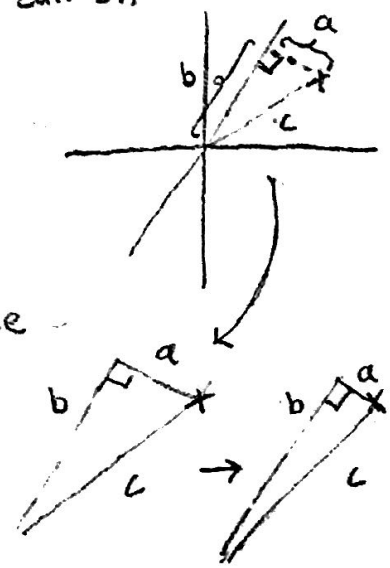
- A, W, and H all have to have non-negative values
- NMF can never undo the application of a latent feature because it cannot use negative values to subtract away the effect
    * thus, it has to be more careful in terms of what it adds at each step
    * in some applications, this allows for more human interpretable latent features
    * though it does lose more info while truncating
- NMF has no orthogonal constraints for latent features so features might be redundant across latent features, but less than LDA and LSA

Principle Component Analysis (PCA)

- unsupervised learning technique that uses an orthogonal transformation to generate new features that are independent of each other
- these new features or axes are called principal components and each principal component tries to account for as much variability in the data as possible i.e largest possible variance
    + principal component 1 (PC1) explains the most variance in the data, PC2 explains the 2nd most variance, and so forth
    + we focus on components that can explain more variance in the data because variance is associated with data separability?
- how does PCA work geometrically in a 2D space for a set of observations defined by 2 features, height and weight?



* We start by plotting our observations by our features after the mean of each feature is subtracted from the data to center the data around the origin so that PC1 can sit along the main axis of the data point cloud
    + in (2), a line through the origin is generated and each data point is orthogonally projected onto the line. The goal of PCA is to minimize the length of this projection, a, since the distance from the origin to the point, c, is fixed, minimizing a is equivalent to maximizing b. If we square b and take the sum across all data points, the goal is to maximize $+\sum_{i=1}^{n} b_i^2$



* we keep rotating the line as in ③ until we find end up with the line with the largest sum of squared distances between the projected points and origin, $\sum_{i=1}^{n} b_i^2$

* this optimal line is called PC1 and the slope of PC1 can be used to find the linear combination of features, height and weight, that generates it.

* If the slope of PC1 is 2, then PC1 = 2·height + 1·weight. This means that the data is mostly spread out among height instead of weight i.e. height offers more data separability

* finding PC2 is easy because it is just orthogonal to PC1. Linearly independent or uncorrelated features are just features that are orthogonal to each other

- how does PCA work for a n-dimensional space?

* a sample covariance matrix for all features is generated

variance measures how a feature varies between itself while covariance measures how two features with each other.

* Sample covariance is used instead of covariance because with a limited sample size, the sample mean might not be close to the actual mean. Dividing by n-1 instead of n in variance helps with avoiding bias

* Sample variance $s_a^2 = \frac{1}{(n-1)} \sum_{i=1}^{n} (a_i - \bar{a})$ variance $\sigma^2 = \frac{1}{(n)} \sum_{i=1}^{n} (a_i - \bar{a})$

average of a ↓

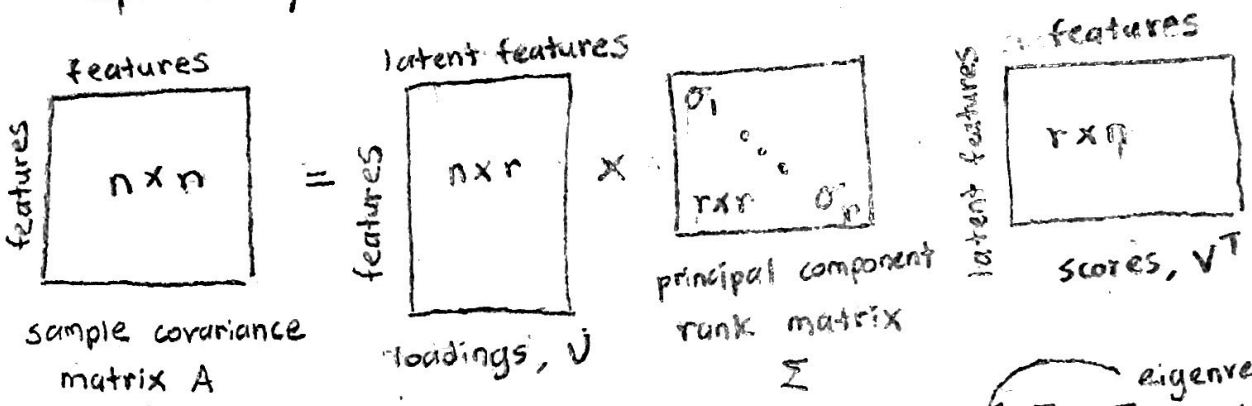
Sample covariance $s_{ab} = \text{cov}(a,b) = \frac{1}{n-1} \sum_{i=1}^{n} (a_i - \bar{a})(b_i - \bar{b})$

$$= \begin{cases} E[(a-\bar{a})(b-\bar{b})] & \text{if } a \neq b, \text{ off-diagonal} \\ E[(a-\bar{a})^2] & \text{if } a = b, \text{ diagonal} \end{cases}$$

Sample covariance matrix A $= \begin{pmatrix} E[(x_1-\mu_1)^2] & \cdots & E[(x_1-\mu_1)(x_p-\mu_p)] \\ \vdots & \ddots & \\ E[(x_p-\mu_p)(x_1-\mu_1)] & & E[(x_p-\mu_p)^2] \end{pmatrix}$

$$= E[(X-\bar{X})(X-\bar{X})^T) = \frac{XX^T}{n-1}$$

* calculating the sample covariance matrix effectively centers the data around the origin
* $\sigma^2_{ab} < 0$ means a and b are negatively correlated.
  $\sigma^2_{ab} = 0$ means a and b are not correlated
  $\sigma^2_{ab} > 0$ means a and b are positively correlated
* SVD is commonly used to decompose the sample covariance matrix A in order to find our principal components because Σ in $U\Sigma V^T$ tells us how much variance can be explained by our latent features i.e. principal components

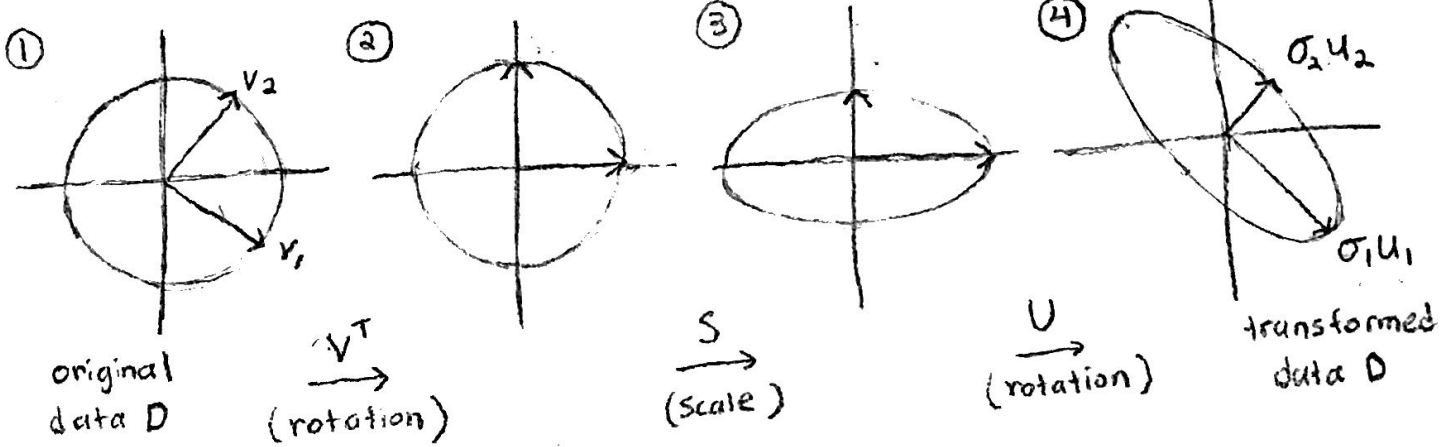


$$A = U\Sigma V^T = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \dots + \sigma_r u_r v_r^T$$

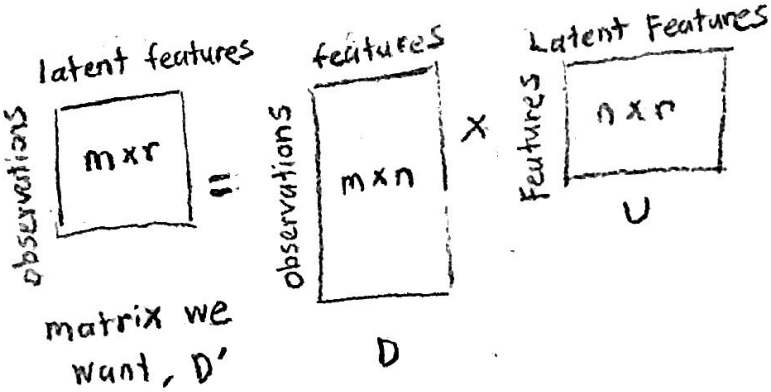
PC1

eigenvalue: weight

eigenvector: direction



* multiplying our original data $D_{m \times n}$ with m observations and n rows by U from SVD of our sample covariance matrix A gives us $D'_{m \times r}$, which shows how principal components map are related to latent features



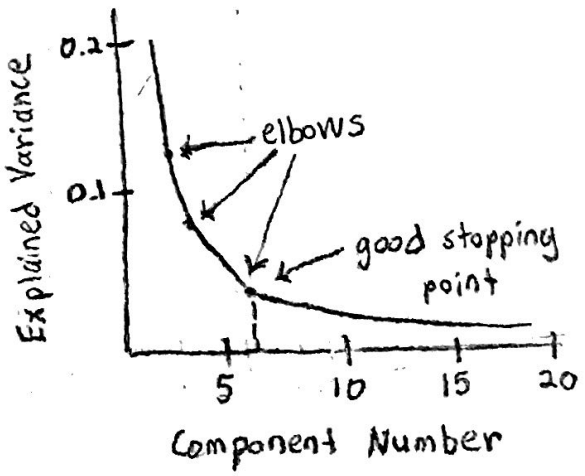
- not always necessary to standardize the data before PCA
    * PCA selects for features with higher variance, so features with larger ranges and scales might cause PCA to just primarily select from these features. Thus, standardize.
- since we do not have labels, the variance PCA detects may not actually be helpful
- PCA only works for linear relationships between features
- components can be tricky to intrepret
- the values in matrix $D'_{m \times r}$ with m observations and r principal components show how observations relate to principal components and are referred to as loading scores
    * a negative sign indicates negative correlation while a positive sign indicates positive correlation
    * the magnitude of the loading scores shows how strongly correlated the documents are to a principal component, and generally regarded as the only thing that matters

Dimensionality Reduction Metrics

Objective

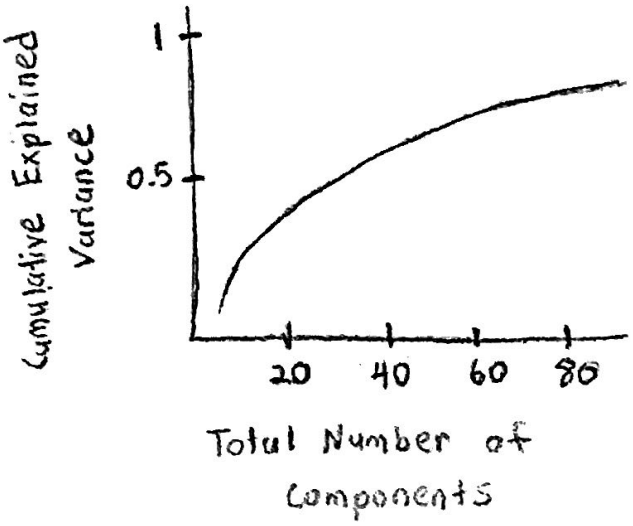
- how do we decide on the optimal number of latent features?
- we can use explained variance since dimensionality reduction seeks to extract features based on variance

Explained Variance vs. Component Number



- a scree plot of explained variance vs component number tells us how much variance each component explains
- an 'elbow' in the plot indicates where there is a relatively large decrease in the amount of explained variance after e.g. 6 components on the left plot before diminishing returns

Cumulative Explained Variance vs. Total Number of Components



- a scree plot of cumulative explained variance vs total number of components tells us how much each additional component contributes to cumulative explained variance
- theoretically, as the number of components approaches ∞, you can reach 100% cumulative explained variance
- while the model might perform better with more components, having too many latent features is not human interpretible or friendly

- use both the explained variance and cumulative explained variance scree plots to decide on a number of components to use before diminishing returns in explained variance is reached
- then decide it using the same number of components or less than that is more interpretable. In the case of text, latent features are topics.

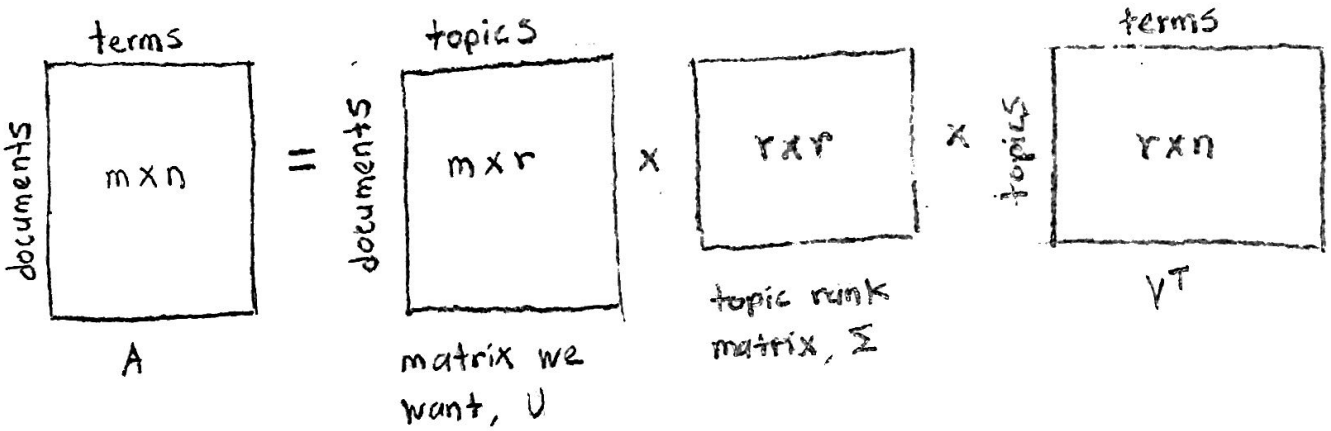
Topic Modeling

Objective

- given a document-term matrix A where the observations are the documents and the features are the terms, what are some latent features i.e the main topics of all the documents?

Latent Semantic Analysis (LSA)

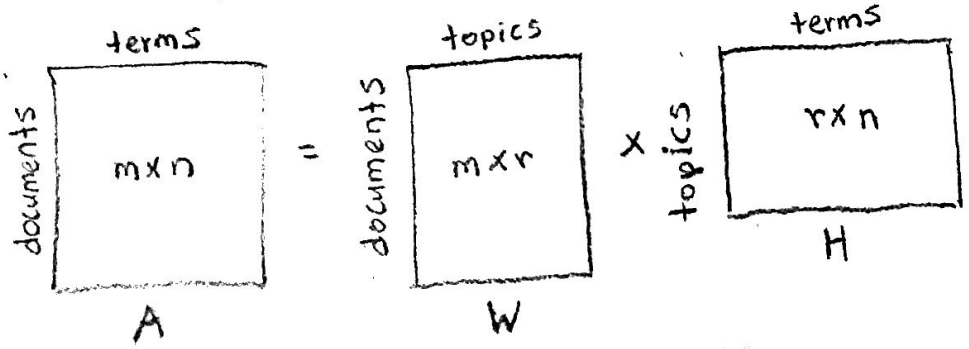
- uses SVD (the sklearn implementation uses truncated version) to produce document-topic matrix



- can be computationally expensive to use since it is performing matrix decomposition, especially with more documents and/or terms
- not possible to update with new document because matrix decomposition requires all documents to be in initial document-term matrix
- a large set of documents and vocabularly is preferred to get more accurate results, especially for more specific text domains
- difficult to say what the topics are
- quick and efficient to use

Non-Negative Matrix Factorization (NMF)

- uses a simpler matrix decomposition to produce document-topic matrix



- NMF works for text since term frequency matrices are non-negative by default i.e. it is not possible to have a negative word count
- may not perform as well as LSA, but tends to have more human interpretable topics
- can be computationally expensive to use since it is performing matrix decomposition, especially with more documents and/or terms
- not possible to update with new document because matrix decomposition requires all documents to be in initial document-term matrix
- quicker and more efficient to use than LSA since only two matrices are computed, which is also why it is more storage efficient than LSA if you have to store all the matrices

Principal Component Analysis (PCA)

- a sample covariance matrix A is computed from the document-term matrix before SVD is used to extract principal components i.e. topics
- since principal components have to be uncorrelated or orthogonal to each other, it is less likely to see the same words in multiple topic groups because the orthogonality constraint does not allow for such linear combinations of terms
- can be computatationally expensive to use since it is performing matrix decomposition, especially with more documents and/or terms
- not possible to update with new document because matrix decomposition requires all documents to be in initial document-term matrix
- good for visualizations if you use PC1, PC2, and even PC3 as axes for plotting documents

Latent Dirichlet Allocation (LDA)

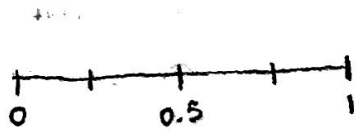
- LDA is a bayesian approach for generating the document-topic matrix. The basic idea is that documents are represented as random mixtures over latent topics, and each topic is characterized by a distribution over words. Dirichet priors are used for the document-topic and term-topic distributions

- What is a dirichlet distribution?

  * it is a probability distribution over a probability simplex for $K \geq 2$

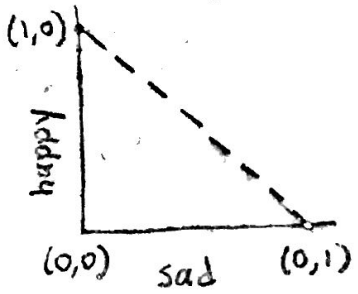
  * given a k-dimensional probability simplex, a simplex is just a way of representing all the possible sets of numbers over k distinct categories. When these sets of numbers are normalized, we now have sets of probabilities that add up to 1 or a probability distribution

$Emotion_1 = \{happy\}$



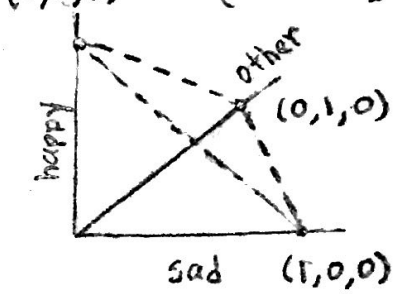
1-D Simplex

$Emotion_2 = \{happy, sad\}$



2-D Simplex

$Emotion_3 = \{happy, sad, other\}$



3-D Simplex

Given 4 topics, we have a 4-D simplex that may have a set like:

$(17, 13, 24, 46) \longrightarrow (0.17, 0.13, 0.24, 0.46)$

probability of topic 1 in document

probability of topic 3 in document

  * essentially, a dirichlet distribution is a distribution over distributions

- we use dirichelet priors because the posterior distribution i.e. what is the probability of a document being topic 0, topic 1, ..., topic n can be represented as a dirichlet distribution

- How does LDA work?

  * let w represent a word. There are N words.

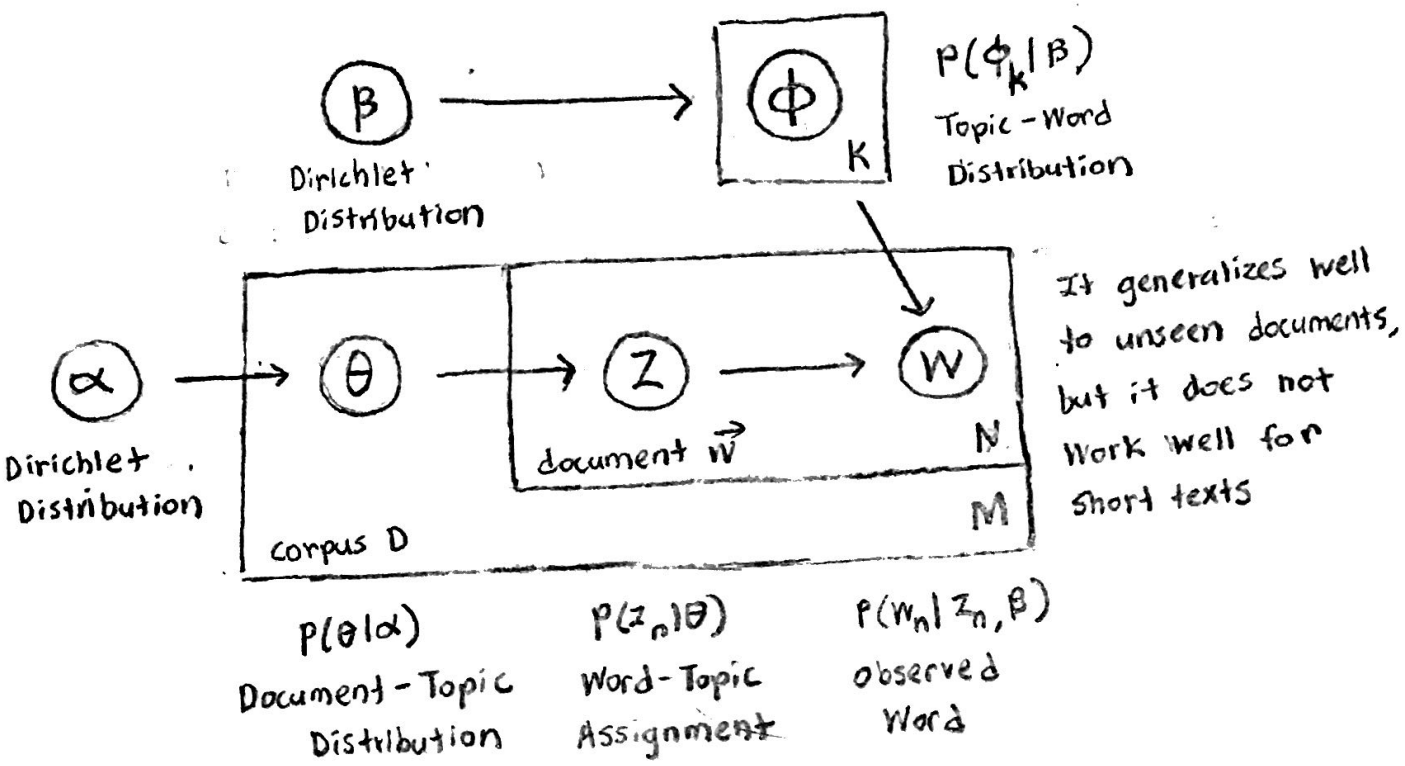
  * let $\vec{w} = (w_1, w_2, \ldots, w_N)$ represent a collection of words i.e. a document There are M documents

  * let $D = \{\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_M\}$ represent a collection of documents i.e. corpus

  * let α be a dirichlet distribution where each set of probabilities is $\theta_{i,K}$ = probability that document $\vec{w} = \{1, \ldots, M\}$ has topic $k \in \{1, \ldots, K\}$

  * let β be a dirichlet distribution where each set of probabilities is $\phi_{k,v}$ = probability of word $w \in \{1, \ldots, N\}$ in topic $k \in \{1, \ldots, K\}$

  * let z be a latent topic. There are K topics



* $p(\vec{w}|\alpha,\beta) = \int p(\theta|\alpha) \left(\prod_{n=1}^{N} \sum_{z_n} p(z_n|\theta)\, p(w_n|z_n,\beta) \right) d\theta$

gives the mariginal distribution of a document i.e. what mix of topics

* $p(D|\alpha,\beta) = \prod_{d=1}^{M} \int p(\theta_d|\alpha) \left(\prod_{n=1}^{N_d} \sum_{z_{dn}} p(z_{dn}|\theta_d)\, p(w_{dn}|z_{dn},\beta) \right) d\theta_d$

$= \prod_{d=1}^{M} p(\vec{w}_d|\alpha,\beta) \longrightarrow \ell(\alpha,\beta) = \sum_{d=1}^{M} \log p(w_d|\alpha,\beta)$

$p(D|\alpha,\beta)$ gives the product of mariginal distributions of single documents i.e. what is the probability of a corpus being like this. If we take the log of both sides of the equation for $p(D|\alpha,\beta)$, we arrive at the log likelihood $\ell(\alpha,\beta)$. The goal is to update α, β to maximize the log likelihood $\ell(\alpha,\beta)$

* all together, we specify how many topics LDA should use. The algorithm will assign every word to a temporary topic, according to a dirichlet distribution. The topic assignment is temporary since it will be updated. For each word in every document, the algorithm will check and update topic assignments based on a) how prevalent that word is across topics i.e. α and b) how prevalent topics are in the document i.e. β to maximize log likelihood $\ell(\alpha,\beta)$

Topic Modeling Metrics

Using Explained Variance

- in most models, dimensionality reduction is used to obtain the topics and which terms are used is based on variance
- using explained variance vs component number and cumulative explained variance vs. total number of components can help you determine how much each additional topic is worth
- while cumulative explained variance increases with the number of topics, having too many topics is not human interpretible or friendly

Topic Coherence Score

- topic coherence is a measure of how similar words within a topic group are to each other
- the assumption is that topics with higher topic coherence scores are better topics. Topic coherence scores can be used to compare topics.
- most methods compute the sum $\sum_{i<j} \text{score}(w_i, w_j)$ of the pairwise scores of the n top words $w_1, \ldots, w_n$ of the topic
- the extrinsic UCI measure uses an external dataset like wikipedia to determine how frequently words w_i and w_j are seen. Let D be a corpus.

$$P(w_i) = \frac{D_{wiki}(w_i)}{D_{wiki}}$$

$D_{wiki}(w_i)$ ← count of documents with word w_i in wiki corpus

D_{wiki} ← total count of documents in wiki corpus

} probability of seeing w_i in a random document in wiki corpus

$$P(w_i, w_j) = \frac{D_{wiki}(w_i, w_j)}{D_{wiki}}$$

$D_{wiki}(w_i, w_j)$ ← count of documents with word w_i and w_j in wiki corpus

} probability of seeing both w_i and w_j in a random document in wiki corpus

$$\text{score}_{UCI}(w_i, w_j) = \log \frac{P(w_i, w_j)}{P(w_i)\,P(w_j)} = \log P(w_j \mid w_i)$$

* you will need a more context-specific external dataset for more domain specific text purposes, which can be difficult to obtain

- the intrinsic UMass measure uses your corpus to determine how frequently words W_i and W_j are seen. Let D be a corpus.

$$P(W_i) = \frac{D(W_i)}{D}$$

$D(W_i)$ ← count of documents with word W_i in corpus

D ← total count of documents in corpus

} probability of seeing W_i in a random document in corpus

$$P(W_i, W_j) = \frac{D(W_i, W_j)}{D}$$

$D(W_i, W_j)$ ← count of documents with word W_i and W_j in corpus

} probability of seeing W_i and W_j in a random document in corpus

$$\text{Score}_{\text{UMass}}(W_i, W_j) = \log P(W_j | W_i) = \log \frac{P(W_i, W_j)}{P(W_i)P(W_j)}$$

$$= \log \frac{D(W_i, W_j)/D}{D(W_i)/D} = \log \frac{D(W_i, W_j)}{D(W_i)}$$

$$\rightarrow \log \frac{D(W_i, W_j) + 1}{D(W_i)}$$ ← add 1 for smoothing

* just like $\text{Score}_{\text{UMass}}(W_i, W_j)$, $\text{Score}_{\text{UCI}}(W_i, W_j)$ is proportional to the probability of seeing W_j given that W_i has been seen.

+ make sure your corpus has enough documents and each document has enough words to obtain a reasonable score

- topic coherence score vs number of topics could be a good plot

Clustering Algorithms

K-Means

- partitioning algorithm that divides data into K clusters, K is user-specified
- points are assigned to a cluster based on metric, typically Euclidean distance, to nearest cluster centroid
- How does k-means work?
  - * randomly initiate K centroids, typically spread out from each other
  - * assign points to cluster based on nearest centroid
  - * recompute centroid values based on points in respective cluster
  - * unassign points from centroids
  - * repeat last 3 steps until algorithm converges, typically based on minimizing inertia score
- strengths:
  - * simple, only need to specify one parameter - K clusters
  - * typically fast, runtime is $O(nkdi)$ for n points, d dimensions, and i iterations
  - * guaranteed to converge
  - * easy to implement
- Weaknesses:
  - * optimal number of clusters K is not obvious
  - * can get trapped in local minima, initial centroid positions matter
  - * sensitive to outliers since centroid values are recomputed based on points within cluster
  - * scaling affects results because Euclidean distance is typically used and it cannot tell apart (0,0,1) and (1,0,0)
- consider using k-modes, an extension of k-means for data that only has categorical data
- consider using k-prototypes, an algorithm that combines k-modes and k-means for data that has both numerical and categorical data

DBSCAN

- clustering algorithm that groups points closely packed together with a certain number of neighboring points

- How does DBSCAN work?
  * it requires 2 parameters, eps and minPoints
  * ε specifies how close points should be to each other to be considered a part of a cluster. If the distance between two points is lower or equal to ε, these points are neighbors
  * minPoints is the minimum number of points to form a dense region i.e. whether points are closely packed together to be clustered
  * for each point, the algorithm checks how many neighboring points a point has. A point is considered reachable from another point if the distance between the two is less than or equal to ε
  * a point not reachable from any other point is an outlier
  * a point with at least minPoints number of reachable points is considered a core point
  * a core point forms a cluster with all points, core or non-core
- Strengths:
  * does not need to specify the number of clusters
  * can find arbitrarily shaped clusters
  * robust to outliers
  * simple, only requires 2 parameters
  * mostly deterministic, usually get the same result
- Weaknesses:
  * not fast, $O(n^2)$ for n points since it may check the same points to see if it is a neighboring point
  * depends heavily on distance metric used
  * might be difficult to pick ε
  * does not work if points are densely packed in some regions but losely packed together in others

Mean-Shift

- gradient based method of finding centroids to produce clusters from. Candidates for centroids are updated to be the mean of points within a given region

- How does Mean Shift Work?
  * start with a centroid at a point
  * sample local density, follow gradient towards denser direction until local density maximum is found
  * designate local density maximum as centroid
  * repeat last 3 steps until all local density maxima/centroids found
  * assign points to centroids, based on euclidean distance
- Strengths:
  * no need to specify any parameters
  * works wells with unevenly packed points
- Weaknesses:
  * slow with a lot of data, but can handle many clusters
  * does not handle arbitrarily shaped clusters
  * euclidean distance only

Spectral Clustering

- graph-based clustering approach that works by non-linearly mapping points to lower dimensional space via a kernel function before applying k-means
- How does spectral clustering work?
  * transform space into fewer dimensions by taking eigenvectors of the point distances matrix
  * perform k-means on lower dimensional space
  * undo space transformation after k-means
- Strengths:
  * best with sparse distances
  * tends to find even sized clusters
  * can handle most arbitrarily shaped clusters e.g. concentric circles
- Weaknesses:
  * does not scale well with more clusters due to low dimensional space
  * Euclidean distance only

Agglomerative Hierarchical Clustering

- bottom up approach to building hierachy of clusters

- How does agglomerative hierarchical clustering work?
  * find closest pair of points and merge into a cluster
  * find next closest pair of points and merge into separate cluster
  * if the next closest pair of points are in different clusters, merge the clusters together
  * repeat until the desired number of clusters is reached or when clusters are too far apart to merge, based on some minimum cluster distance threshold
- different ways to link:
  * 'single', distance between two clusters is minimum pairwise distances
    Fast, but can form long chains instead of clusters
  * 'complete', distance between two clusters is maximum pairwise distance
    Fast, no chains, sensitive to outliers
  * 'average', distance between two clusters is average of all pairwise distance
    Slower, no chains, not sensitive to outliers
  * 'ward', merge two clusters based on which combination gives best score
    i.e. lowest inertia and in turn, minimum within cluster variance or most tightly packed clusters. Same as Ward
- Strengths:
  * full hierarchy tree, useful if you want different levels of granularity
  * lots of distance metric and linkage options
- Weaknesses:
  * slow, $O(n^2)$ due to cluster merging
  * finds uneven cluster sizes due to how chaining occurs

Distance Metrics

- different options to consider for clustering
  * Euclidean
  * Manhattan
  * Minkowski
  * Cosine Similarity
  * Jaccard Similarity
  * Jensen-Shannon Divergence

Clustering Algorithm Metrics

Inertia

- inertia is a measure of how close points are to their respective centroid
- the assumption behind inertia is that good clustering is defined by points being close to their respective centroid
- thus, a smaller inertia value is better, with the minimum value being 0. However, a score of 0 is misleading and useless if:
  * all points are in the exact same location.
  * the number of clusters is the same as the number of points
- the goal is to minimize the sum of squares of distances of points with its respective cluster centroid
- the distance metric typically used is euclidean distance, but consider using a different metric if the clustering algorithm is not based on euclidean distance

$$I = \sum_{j=1} \sum_{x_i \in \text{cluster } j} |x_i - x_{\text{centroid},j}|^2$$

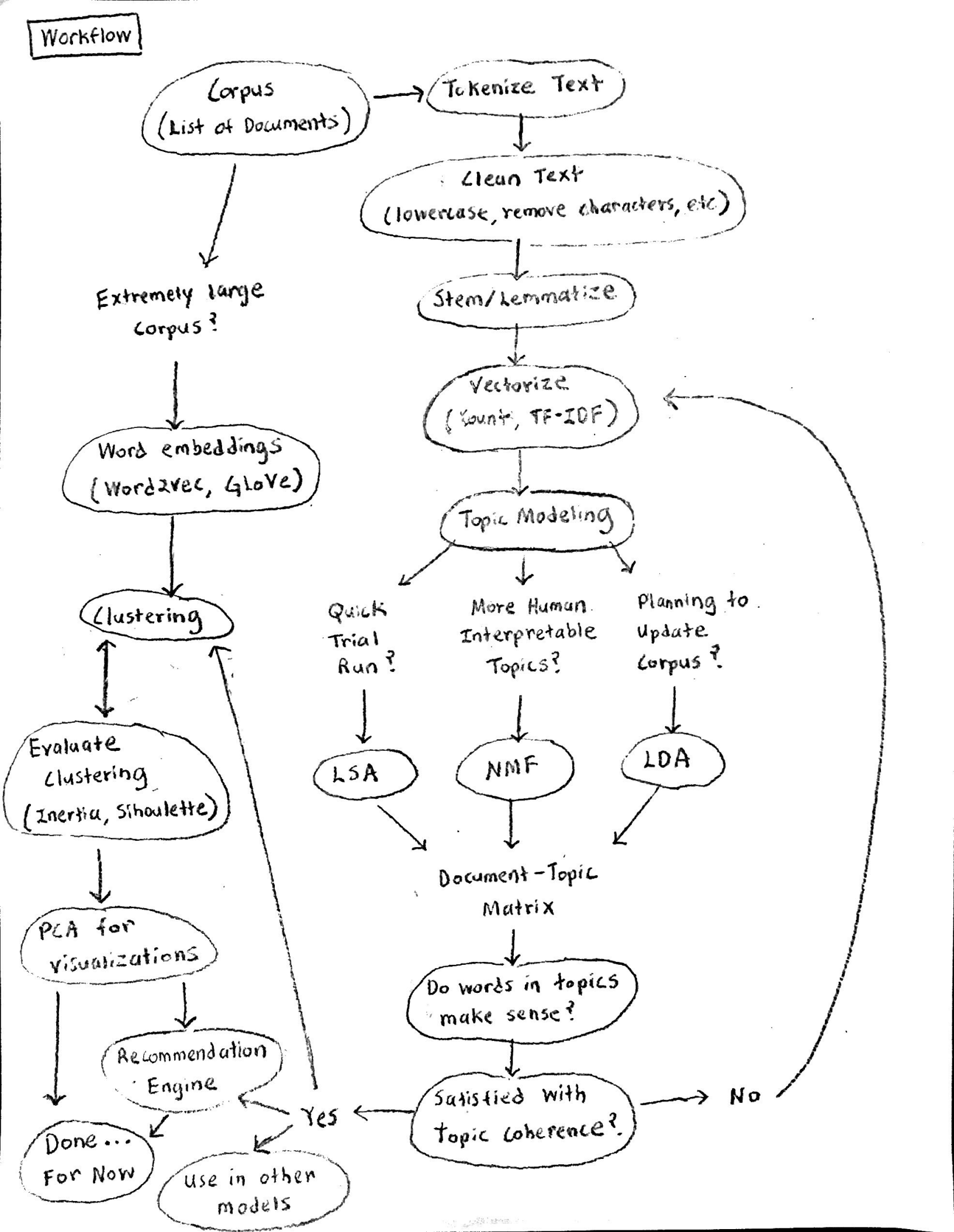
- sum over clusters
- sum over points in cluster j
- position of point i in cluster j
- position of centroid of cluster j

Sihoulette Coefficient

- sihoulette score is a measure of how close points are to their respective centroid in respect to other centroids
- the assumption behind using sihoulette coefficient is that good clustering is defined by points being closed to their respective centroid and far away from other centroids
- let $a(i)$ be the mean distance between i and all other points in same cluster
- let $b(i)$ be the mean distance between i and all other points in nearest cluster that does not include i

$$s(i) = \begin{cases} 1 - a(i)/b(i) & \text{if } a(i) < b(i) \\ 0 & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1 & \text{if } a(i) > b(i) \end{cases}$$

- the distance metric typically used is euclidean distance, but consider using a different metric if the clustering algorithm is not based on euclidean distance


Workflow
Corpus (List of Documents)
Tokenize Text
Clean Text (lowercase, remove characters, etc)
Stem/Lemmatize
Vectorize (Count, TF-IDF)
Topic Modeling
Quick Trial Run?
More Human Interpretable Topics?
Planning to Update Corpus?
LSA
NMF
LDA
Document-Topic Matrix
Do words in topics make sense?
Satisfied with Topic Coherence?
No
Yes
Use in other models
Extremely large corpus?
Word embeddings (Word2vec, GloVe)
Clustering
Evaluate Clustering (Inertia, Silhoulette)
PCA for visualizations
Recommendation Engine
Done... For Now