Computational Sociology

NLP II: Word Embeddings

Dr. Thomas Davidson

Rutgers University

March 8, 2021

Plan

1. Course updates
2. Words and texts as vectors
3. The vector-space model review
4. Latent semantic analysis
5. Interlude: Language models
6. Word embeddings
7. Contextualized embeddings

Course updates

- Spring break next week, no class

The vector-space model review

Vector representations

- Last week we looked at how we can represent texts as numeric vectors
  - Documents as vectors of words
  - Words as vectors of documents
- A document-term matrix (DTM) is a matrix where documents are represented as rows and tokens as columns

The vector-space model review

Weighting schemes

- We can use different schemes to weight these vectors
    - Binary (Does word w_i occur in document d_j?)
    - Counts (How many times does word w_i occur in document d_j?)
    - TF-IDF (How many times does word w_i occur in document d_j, accounting for how often w_i occurs across all documents $d \in D$?)
        - Recall *Zipf's Law*: a handful of words account for most words used; such words do little to help us to distinguish between documents

The vector-space model review

Cosine similarity

$$cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \|\vec{v}\|} = \frac{\sum_i \vec{u}_i \vec{v}_i}{\sqrt{\sum_i \vec{u}_i^2} \sqrt{\sum_i \vec{v}_i^2}}$$

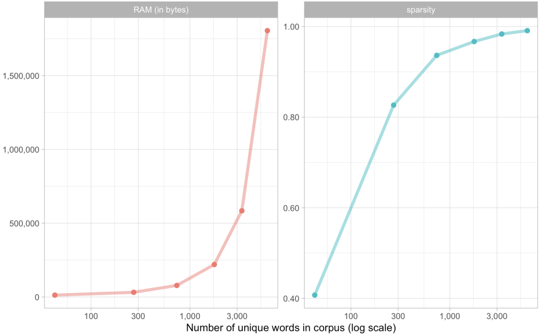
The vector-space model review

Limitations

- These methods produce *sparse* vector representations
  - Given a vocabulary of unique tokens V, each vector contains $|V|$ elements.
  - Most values in a DTM are zero.
- This is computationally inefficient, since most entries in a DTM are equal to zero

The vector-space model review

Limitations



Source: https://smltar.com/embeddings.html

Latent semantic analysis

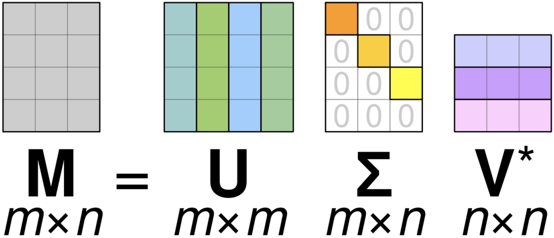
Latent Semantic Analysis

- One approach to reduce dimensionality and better capture semantics is called **Latent Semantic Analysis** (*LSA*)
  - We can use a process called *singular value decomposition* to find a *low-rank approximation* of a DTM.
    - In short, we can "squash" a big matrix into a much smaller matrix while retaining important information.

$$DTM = X = U\Sigma V^T$$

Latent semantic analysis

Singular Value Decomposition



See the Wikipedia page for video of the latent dimensions in a sparse TDM.

Latent semantic analysis

Example: Shakespeare's writings

X is a TF-IDF weighted Document-Term Matrix of Shakespeare's writings from Project Gutenberg. There are 11,666 unique tokens (each of which occurs 10 or more times in the corpus) and 66 documents.

```
X <- as.matrix(read.table("shakespeare.txt"))
X <- X[, which(colSums(X) != 0)] # Drop zero columns

dim(X)
```

```
## [1]    66 11666
```

Latent semantic analysis

Creating a lookup dictionary

We can construct a list to allow us to easily find the index of a particular token.

```
lookup.index.from.token <- list()
for (i in 1:length(colnames(X))) {
  lookup.index.from.token[colnames(X)[i]] <- i
}
```

Latent semantic analysis

Using the lookup dictionary

This easily allows us to find the vector representation of a particular word. Note how most values are zero since the character Hamlet is only mentioned in a handful of documents.

```
lookup.index.from.token["hamlet"]
```

```
## $hamlet
## [1] 8231
```

```
round(as.numeric(X[,unlist(lookup.index.from.token["hamlet"])]),3)
```

```
##  [1] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.0
## [13] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.0
## [25] 0.046 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.0
## [37] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.0
## [49] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.010 0.000 0.000 0.0
## [61] 0.014 0.000 0.000 0.002 0.000 0.000
```

Latent semantic analysis

Calculating similarties

The following code normalizes each column and constructs a word-word cosine-similarity matrix.

```
normalize <- function(X) {
  for (i in 1:dim(X)[2]) {
    X[,i] <- (X[,i]/sqrt(sum(X[,i]^2)))
  }
  return(X)
}

X.n <- normalize(X)

sims <- t(X.n) %*% X.n
dim(sims)
```

```
## [1] 11666 11666
```

Latent semantic analysis

Most similar function

For a given token, this function allows us to find the `n` most similar tokens in the similarity matrix, where `n` defaults to 10.

```
get.top.n <- function(token, sims, n=10) {
  top <- sort(sims[unlist(lookup.index.from.token[token]),],
                  decreasing=T)[1:n]
  return(top)
}
```

Latent semantic analysis

Finding similar words

```r
get.top.n("love",sims)
```

```
##      love      fair       lie     music     sight    beauty    breat
## 1.0000000 0.8583652 0.8533865 0.8425000 0.8213188 0.8098769 0.796472
##     shine      dead
## 0.7616598 0.7616076
```

```r
get.top.n("hate", sims)
```

```
##      hate   flatter     happy     power      time    forgot       pas
## 1.0000000 0.7925268 0.7785826 0.7709535 0.7473243 0.7378957 0.734621
##   friends      kill
## 0.7310338 0.7300051
```

```r
get.top.n("romeo", sims)
```

```
##     romeo  mercutio    tybalt   tybalts   capulet  benvolio montague
## 1.0000000 0.9999330 0.9999318 0.9998235 0.9995140 0.9992019 0.998216
##   sampson    juliet
## 0.9923716 0.9906526
```

Latent semantic analysis

Singular value decomposition

The `svd` function allows us to decompose the DTM. We can then easily reconstruct it using the formula shown above.

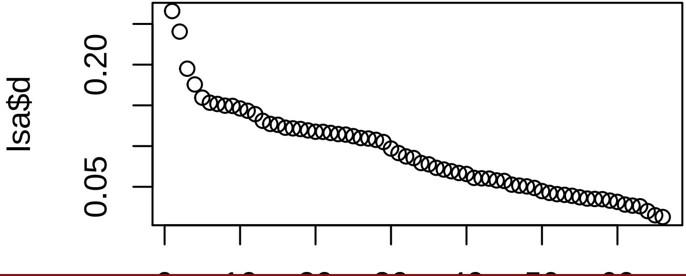
```
# Computing the singular value decomposition
lsa <- svd(X)

# We can easily recover the original matrix from this representation
X.2 <- lsa$u %*% diag(lsa$d) %*% t(lsa$v) # X = U \Sigma V^T

# Verifying that values are the same, example of first column
sum(round(X-X.2,5))
```

```
## [1] 0
```

Latent semantic analysis

Singular value decomposition

This plot shows the magnitude of the singular values (the diagonal entries of Σ). Roughly speaking, the magnitude of the singular value corresponds to the amount of variance explained in the original matrix.



Latent semantic analysis

Truncated singular value decomposition

In the example above retained the original matrix dimensions. The point of latent semantic analysis is to compute a *truncated* SVD such that we have a new matrix in a sub-space of X. In this case we only want to retain the first two dimensions of the matrix.

```
k <- 15 # Dimensions in truncated matrix

# We can take the SVD of X but only retain the first k singular values
lsa.2 <- svd(X, nu=k, nv=k)

# In this case we reconstruct X just using the first k singular values
X.trunc <- lsa.2$u %*% diag(lsa.2$d[1:k]) %*% t(lsa.2$v)

# But the values will be slightly different since it is an approximatio
sum(round(X-X.trunc,5))
```

```
## [1] 15.35817
```

Latent semantic analysis

Recalculating similarties using the LSA matrix

```
words.lsa <- t(lsa.2$v)
colnames(words.lsa) <- colnames(X)

round(as.numeric(words.lsa[,unlist(lookup.index.from.token["hamlet"])])
```

```
##  [1]  0.000  0.074 -0.003  0.001 -0.005  0.028 -0.035 -0.010  0.022
## [11]  0.006 -0.010  0.049 -0.006 -0.003
```

Latent semantic analysis

Recalculating similarties using the LSA matrix

```
words.lsa.n <- normalize(words.lsa)
sims.lsa <- t(words.lsa.n) %*% words.lsa.n
```

Latent semantic analysis

Comparing similarities

```
get.top.n("love",sims)
```

```
##      love      fair       lie     music     sight    beauty    breat
## 1.0000000 0.8583652 0.8533865 0.8425000 0.8213188 0.8098769 0.796472
##     shine      dead
## 0.7616598 0.7616076
```

```
get.top.n("love",sims.lsa)
```

```
##      love   counsel     loves    tears   modesty    loving       fai
## 1.0000000 0.9514839 0.9406690 0.9385571 0.9377943 0.9318378 0.929927
##     oaths  forsworn
## 0.9141528 0.9133888
```

Latent semantic analysis

Comparing similarities

```
get.top.n("hate", sims)
```

```
##      hate   flatter     happy     power      time    forgot       pas
## 1.0000000 0.7925268 0.7785826 0.7709535 0.7473243 0.7378957 0.734621
##   friends      kill
## 0.7310338 0.7300051
```

```
get.top.n("hate", sims.lsa)
```

```
##      hate miserable     anger       art      bare    breath     fault
## 1.0000000 0.9697931 0.9683188 0.9409666 0.9401336 0.9303231 0.929732
##      aged      thee
## 0.9285337 0.9275189
```

Latent semantic analysis

Comparing similarities

```
get.top.n("romeo", sims)
```

```
##     romeo  mercutio    tybalt   tybalts   capulet  benvolio montague
## 1.0000000 0.9999330 0.9999318 0.9998235 0.9995140 0.9992019 0.998216
##   sampson    juliet
## 0.9923716 0.9906526
```

```
get.top.n("romeo", sims.lsa)
```

```
##     romeo montagues   tybalts    tybalt  mercutio mercutios   capule
## 1.0000000 0.9999961 0.9999937 0.9999933 0.9999923 0.9999910 0.999988
##  capulets    romeos
## 0.9999846 0.9999741
```

Latent semantic analysis

Comparing similarities

```
get.top.n("hamlet", sims)
```

```
##    hamlet   horatio marcellus   ophelia  polonius  barnardo    laerte
## 1.0000000 0.9829677 0.9824643 0.9607921 0.9600178 0.9591448 0.958729
## voltemand  lucianus
## 0.9465517 0.9308989
```

```
get.top.n("hamlet", sims.lsa)
```

```
##    hamlet  gertrude    danish   pyrrhus   denmark wittingly      polo
## 1.0000000 0.9981001 0.9980634 0.9962985 0.9962098 0.9961391 0.996088
##      laer    norwey
## 0.9960141 0.9959639
```

Latent semantic analysis

Execise
Re-run the code above with a different value of `k` on line 188. Compare some terms in the original similarity matrix and the new matrix. How does changing `k` affect the results?

```
get.top.n("", sims)
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA
```

```
get.top.n("", sims.lsa)
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA
```

Latent semantic analysis

Inspecting the latent dimensions

We can analyze the meaning of the latent dimensions by looking at the terms with the highest weights in each row. In this case I use the raw LSA matrix without normalizing it. In this case the latent dimensions seem to correspond to different plays. This isn't too surprising since the each document was a separate play. These dimensions will be more interesting with larger corpora.

```
for (i in 1:dim(words.lsa)[1]) {
  top.words <- sort(words.lsa[i,], decreasing=T)[1:5]
  print(paste(c("Dimension: ",i), collapse=" "))
  print(top.words)
}
```

```
## [1] "Dimension:  1"
##           amy         bened          bero         botes          cas
## -1.204978e-06 -1.204978e-06 -1.204978e-06 -1.204978e-06 -1.204978e-0
## [1] "Dimension:  2"
##   sidenote   footnote        ham     hamlet       haue
## 0.75168642 0.58659055 0.20541499 0.07354514 0.06784205
```

Latent semantic analysis

Limitations of Latent Semantic Analysis

- Bag-of-words assumptions and document-level word associations
  - We still treat words as belonging to documents and lack finer context about their relationships
    - Although we could theoretically treat smaller units like sentences as documents
- Matrix computations become intractable with large corpora
- A neat linear algebra trick, but no underlying language model

Interlude: Language models

Intuition

- A language model is a probabilistic model of language use
- Given some string of tokens, what is the most likely token?
  - Examples
    - Auto-complete
    - Google search completion

Interlude: Language models

Bigram models

- $P(w_i|w_{i-1})$ = What is the probability of some word w_i given the last word, w_{i-1}?
  - $P(Jersey|New)$
  - $P(Brunswick|New)$
  - $P(York|New)$
  - $P(Sociology|New)$

Interlude: Language models

Bigram models

- We use a corpus of text to calculate these probabilities by studying word co-occurrence.
  - $P(Jersey|New) = \frac{C(New\ \ Jersey)}{C(New)}$, e.g. proportion of times "New" is followed by "Jersey", where $C()$ is the count operator.
- More frequently occurring pairs will have a higher probability.
  - We might expect that $P(York|New) > P(Jersey|New) > P(Brunswick|New) >> P(Sociology|New)$

Interlude: Language models

Incorporating more information

- We can also model the probability of a word, given a sequence of words
- $P(x|S) =$ What is the probability of some word x given a partial sentence S?
- $A = P(Jersey|Rutgers\ University\ is\ in\ New)$
- $B = P(Brunswick|Rutgers\ University\ is\ in\ New)$
- $C = P(York|Rutgers\ University\ is\ in\ New)$
- In this case we have more information, so "York" is less likely to be the next word. Hence,
  - $A \approx B > C$

Interlude: Language models

Estimation

We can compute the probability of an entire sequence of words by using considering the joint conditional probabilities of each pair of words in the sequence. For a sequence of n words, we want to know the joint probability of $P(w_1, w_2, w_3, ..., w_n)$. We can simplify this using the chain rule of probability:

$$P(w_{1:n}) = P(w_1)P(w_2|w_1)P(w_3|w_{1:2})...P(w_n|w_{1:n-1})$$

$$= \prod_{k=1}^{n} P(w_k|w_{1:k-1})$$

Interlude: Language models

Estimation

The bigram model simplifies this by assuming it is a first-order Markov process, such that the probability w_k only depends on the previous word, w_{k-1}.

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k | w_{k-1})$$

These probabilities can be estimated by using Maximum Likelihood Estimation on a corpus.

See https://web.stanford.edu/~jurafsky/slp3/3.pdf for an excellent review of language models

Language models

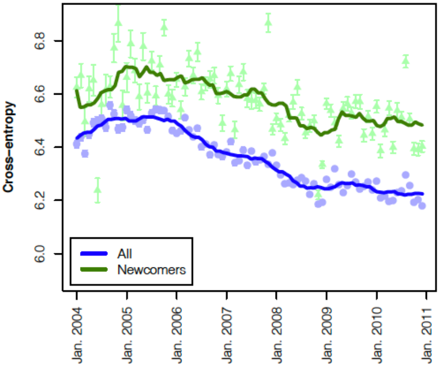
Empirical applications

- Danescu-Niculescu-Mizil et al. 2013 construct a bigram language model for each month on *BeerAdvocate* and *RateBeer* to capture the language of the community
  - For any given comment or user, they can then use a measure called *cross-entropy* to calculate how "surprising" the text is given the language model
- The theory is that new users will take time to assimilate into the linguistic norms of the community

https://en.wikipedia.org/wiki/Cross_entropy

Language models

Empirical applications



(a) BeerAdvocate

Danescu-Niculescu-Mizil, Cristian, Robert West, Dan Jurafsky, Jure Leskovec, and Christopher Potts. 2013. "No Country for Old Members: User Lifecycle and Linguistic Change in Online Communities." In Proceedings of the 22nd International Conference on World Wide Web, 307–18. ACM. http://dl.acm.org/citation.cfm?id=2488416.

Language models

Neural language models

- Recent advances in both the availability of large corpora of text *and* the development of neural network models have resulted in new ways of computing language models.
- By using machine-learning techniques, particularly neural networks, to train a language model, we can construct better vector representations.

Word embeddings

Intuition

- We use the context in which a word occurs to train a language model
  - The model learns by viewing millions of short snippets of text (e.g 5-grams)
- This model outputs a vector representation of each word in k-dimensional space, where $k << |V|$.
  - Like LSA, these vectors are *dense*
    - Each element contains a real number and can be positive or negative

Word embeddings

Word2vec: Skip-gram and continuous bag-of-words (CBOW)

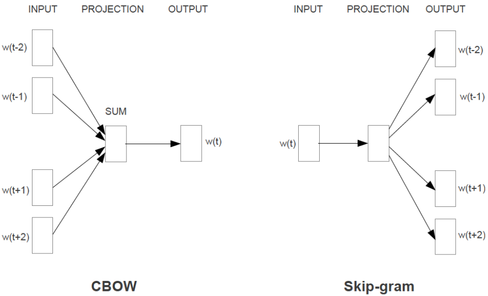


Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

Word embeddings

Word2vec: CBOW intuition

- We start with a string where the focal word is known, but hidden from the model, but we know the context within a window, in this case two words on either side of the focal word
  - e.g. "The cat ? on the", where ? = "sat"
- The model is trained using a process called *negative sampling*, where it must distinguish between the true sentence and "fake" sentences where ? is replaced with another token.
  - Each "guess" allows the model to begin to learn the correct answer
- By repeating this for millions of text snippets the model is able to "learn" which words go with which contexts

Word embeddings

Word2vec: Skip-gram intuition

- We start with a string where the focal is known, but the context within the window is hidden
  - e.g. "$?_1$ $?_2$ sat $?_3$ $?_4$"
- The model tests different words in the vocabulary to predict the missing context words
  - Each "guess" allows the model to begin to learn the correct answer
- By repeating this for millions of text snippets the model is able to "learn" which contexts go with which words

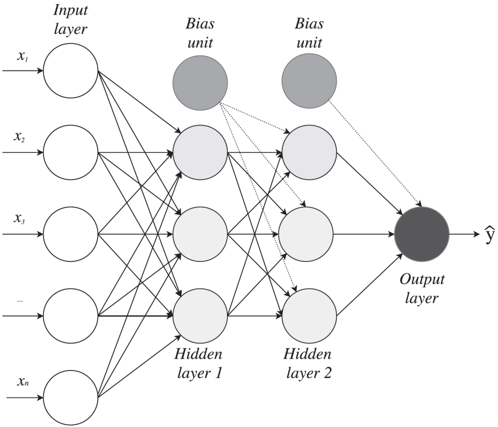
Word embeddings

Word2vec: Model

- Word2vec uses a shallow neural-network to predict a word given a context (CBOW) or a context given a word (skip-gram)
  - But we do not care about the prediction itself, only the *weights* the model learns
- It is a self-supervised method since the model is able to update using the correct answers
  - e.g. In CBOW the model knows when the prediction is wrong and updates the weights accordingly

Word embeddings

Word2vec: Feed-forward neural network



This example shows a two-layer feed-forward neural network.

Word embeddings

Word2vec: Estimation procedure

- Batches of strings are passed through the network
  - After each batch, weights are updated using *back-propagation*
    - The model updates its weights in the direction of the correct answer (the objective is to improve predictive accuracy)
    - Optimization via *stochastic gradient descent*

Word embeddings

Vector representations of words

- Each word is represented as a vector of weights learned by the neural network
  - Each element of this vector represents how strongly the word activates a neuron in the hidden layer of the network
  - This represents the association between the word and a given dimension in semantic space

Word embeddings

Distributional semantics

- The word vectors in the embedding space capture information about the context in which words are used
  - Words with similar meanings are situated close together in the embedding space
- This is consistent with Ludwig Wittgenstein's *use theory of meaning*
  - "the meaning of a word is its use in the language", *Philosophical Investigations* (1953)
- *Distributional semantics* is the theory that the meaning of a word is derived from its context in language use
  - "You shall know a word by the company it keeps", J.R. Firth (1957)

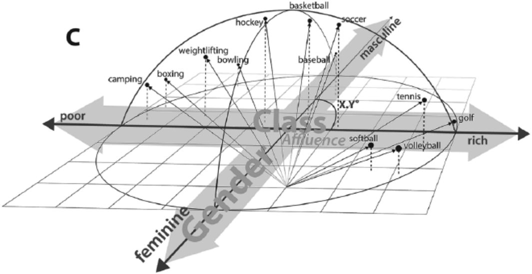
Word embeddings

Analogies

- The most famous result from the initial word embedding paper is the ability of these vectors to capture analogies:
  - $king - man + woman \approx queen$
  - $Madrid - Spain + France \approx Paris$

Word embeddings

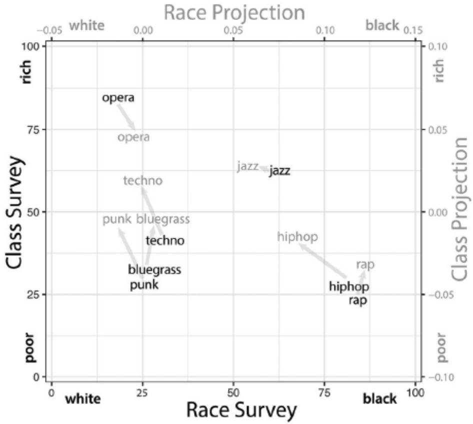
Applications: Understanding social class



Kozlowski, Austin C., Matt Taddy, and James A. Evans. 2019. "The Geometry of Culture: Analyzing the Meanings of Class through Word Embeddings." American Sociological Review, September, 000312241987713. https://doi.org/10.1177/0003122419877135.

Word embeddings

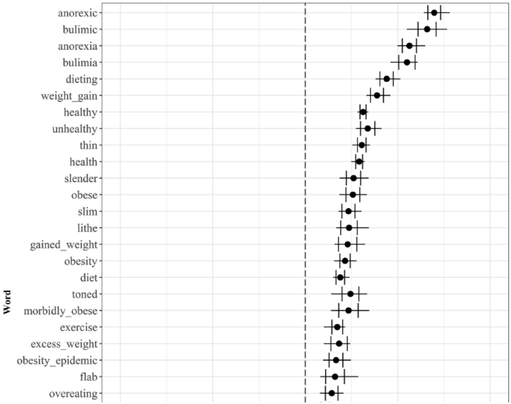
Applications: Understanding social class



Word embeddings

Applications: Understanding cultural schematas

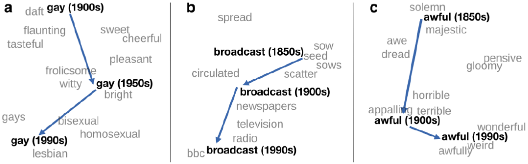
Figure 4: Gendering of Obesity-Related Words



Arseniev-Koehler, Alina, and Jacob G. Foster. 2020. "Machine Learning as a Model for Cultural Learning: Teaching an Algorithm What It Means to Be Fat." Preprint. *SocArXiv*. https://doi.org/10.31235/osf.io/c9yj3.

Word embeddings

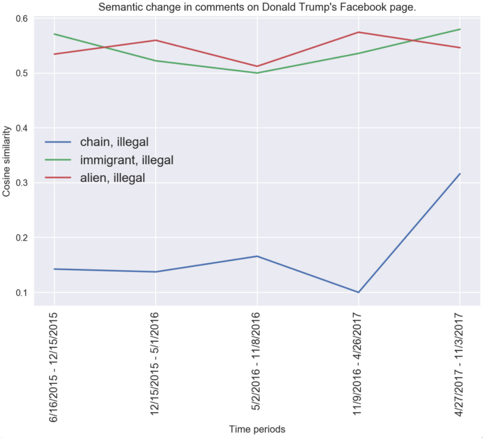
Applications: Semantic change



Hamilton, William L., Jure Leskovec, and Dan Jurafsky. 2016. "Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change." In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 1489–1501.

Word embeddings

Applications: Semantic change



Davidson ~2017, unpublished.

Word embeddings

Pre-trained word embeddings

- In addition to `word2vec` there are several other popular variants including `GloVe` and `Fasttext`
  - Pre-trained embeddings are available to download so you don't necessarily need to train your own
- When to train your own embeddings?
  - You have a large corpus of text ($>$ tens of thousands of documents)
  - You think the underlying language model / data generating process may differ from that represented by existing corpora
    - e.g. A word embedding trained on newspapers may not be very useful for studying Twitter since online language use differs substantially from written news media

Word embeddings

Loading a corpus

```
library(janeaustenr)
library(dplyr)
library(stringr)

original_books <- austen_books() %>%
  group_by(book) %>%
  mutate(linenumber = row_number(),
         chapter = cumsum(str_detect(text,
                                     regex("^chapter [\\divxlc]",
                                           ignore_case = TRUE)))) %>%
  ungroup()
```

Code from https://www.tidytextmining.com/tidytext.html

Word embeddings

Word embeddings in R

We're going to use the library `word2vec` to load a pre-trained word embedding model into R. The library is a R wrapper around a `C++11` library. The the original library can be found here and the R version wrapper here.

```
#install.packages("word2vec")
library(word2vec)
set.seed(987654321) # random seed

model <- word2vec(x = original_books$text,
                  type="cbow",
                  dim=300,
                  window = 10L)
```

Word embeddings

Getting embeddings for words

We can use the `predict` function to find the nearest words to a given term.

```
predict(model, c("love"), type = "nearest", top_n = 10)
```

```
## $love
##    term1    term2 similarity rank
## 1   love   favour  0.7830145    1
## 2   love   really  0.7737371    2
## 3   love   regard  0.7711758    3
## 4   love sensible  0.7613294    4
## 5   love  opinion  0.7595763    5
## 6   love     life  0.7539126    6
## 7   love attached  0.7442610    7
## 8   love    power  0.7422375    8
## 9   love    vouch  0.7402475    9
## 10  love   credit  0.7391796   10
```

Word embeddings

Getting embeddings for words

We can also get the embedding matrix and try to do reasoning by analogy. We can see the model doesn't perform very well. This is because it has only been trained on a small corpus of text.

```
emb <- as.matrix(model)
vector <- emb["King", ] - emb["man", ] + emb["woman", ]
predict(model, vector, type = "nearest", top_n = 10)
```

```
##          term similarity rank
## 1      Steele  0.9979067    1
## 2     Fairfax  0.9942177    2
## 3      Morton  0.9860873    3
## 4        Grey  0.9848179    4
## 5    Carteret  0.9817026    5
## 6      Taylor  0.9793696    6
## 7     Hawkins  0.9749748    7
## 8       Price  0.9620643    8
## 9       Owens  0.9604247    9
## 10    Andrews  0.9598454   10
```

Word embeddings

Getting embeddings for words

Let's try another example.

```r
emb <- as.matrix(model)
vector <- emb["queen", ] - emb["woman", ] + emb["man", ]
predict(model, vector, type = "nearest", top_n = 10)
```

```
##          term similarity rank
## 1       queen  0.9958631    1
## 2       south  0.8930964    2
## 3        vary  0.8888741    3
## 4    ordained  0.8877603    4
## 5       sloop  0.8806481    5
## 6      Amelia  0.8763664    6
## 7     sermons  0.8760002    7
## 8       cloud  0.8700769    8
## 9         Say  0.8676408    9
## 10  imagining  0.8644975   10
```

Word embeddings

Exercise

Modify the parameters of the word embedding algorithm and see if the results improve.

```
### Code here or modify examples above
```

Word embeddings

Loading a pre-trained embedding

Let's try another example. I downloaded a pre-trained word embedding model trained on a much larger corpus of English texts. The file is 833MB in size. Following the documentation we can load this model into R.

```
model.pt <- read.word2vec(file = "../data/sg_ns_500_10.w2v", normalize
```

Word embeddings

Similarities

Find the top 10 most similar terms to “love” in the embedding space.

```
predict(model.pt, c("love"), type = "nearest", top_n = 10)
```

```
## $love
##    term1      term2 similarity rank
## 1   love      loves  0.8190995    1
## 2   love    romance  0.7833382    2
## 3   love     loving  0.7823190    3
## 4   love      adore  0.7729287    4
## 5   love      loved  0.7710815    5
## 6   love      love,  0.7661433    6
## 7   love @ellenpage  0.7653358    7
## 8   love      love/  0.7651593    8
## 9   love    longing  0.7627270    9
## 10  love      free/  0.7627175   10
```

Word embeddings

Similarities

Find the top 10 most similar terms to "hamlet" in the embedding space.

```
predict(model.pt, c("hamlet"), type = "nearest", top_n = 10)
```

```
## $hamlet
##     term1          term2 similarity rank
## 1  hamlet        laertes  0.7913417    1
## 2  hamlet     fortinbras  0.7826205    2
## 3  hamlet    shakespeare  0.7587951    3
## 4  hamlet          osric  0.7433756    4
## 5  hamlet       polonius  0.7390884    5
## 6  hamlet shakespearean  0.7336283    6
## 7  hamlet        village  0.7227796    7
## 8  hamlet    rosencrantz  0.7161442    8
## 9  hamlet       kronborg  0.7136022    9
## 10 hamlet        othello  0.7113878   10
```

Word embeddings

Re-trying the analogy test

Let's re-try the analgy test. We still don't go great but now queen is in the top 5 results.

```
emb <- as.matrix(model.pt)
vector <- emb["king", ] - emb["man", ] + emb["woman", ]
predict(model.pt, vector, type = "nearest", top_n = 10)
```

```
##            term similarity rank
## 1          king  0.9663149    1
## 2        alveda  0.7624112    2
## 3     leonowens  0.7437726    3
## 4       sobhuza  0.7369328    4
## 5         queen  0.7323185    5
## 6        chakri  0.7305732    6
## 7   chulabhorn  0.7290710    7
## 8   sirindhorn  0.7239375    8
## 9        khesar  0.7216632    9
## 10      namgyel  0.7201231   10
```

Word embeddings

Re-trying the analogy test

Let's try another analogy. The "correct' answer is second. Not bad.

```r
vector <- emb["madrid", ] - emb["spain", ] + emb["france", ]
predict(model.pt, vector, type = "nearest", top_n = 10)
```

```
##                term similarity rank
## 1            france  0.9215138    1
## 2             paris  0.9075408    2
## 3            madrid  0.8657743    3
## 4         marseille  0.8486081    4
## 5          charléty  0.8464751    5
## 6          nicollin  0.8420396    6
## 7  superpuissances  0.8416948    7
## 8          moustoir  0.8402181    8
## 9          surplace  0.8379595    9
## 10           juvisy  0.8375083   10
```

Word embeddings

Re-trying the analogy test

Let's try another slightly more complex analogy. Not bad overall.

```
vector <- (emb["new", ] + emb["jersey", ])/2 - emb["trenton", ] + emb["
predict(model.pt, vector, type = "nearest", top_n = 10)
```

```
##         term similarity rank
## 1     albany  0.9643639    1
## 2        new  0.9062052    2
## 3       york  0.8031820    3
## 4    mceneny  0.7575994    4
## 5    upstate  0.7556026    5
## 6       wgdj  0.7479031    6
## 7      cuomo  0.7402880    7
## 8     jersey  0.7357954    8
## 9     brunos  0.7214818    9
## 10    panynj  0.7166725   10
```

Word embeddings

Representing documents

Last week we focused on how we could represent documents using the rows in the DTM. So far we have just considered how words are represented in the embedding space. We can represent a document by summing over the vectors and taking the average vector:

```
descartes <- (emb["i", ] +
                emb["think", ] +
                emb["therefore", ] +
                emb["i", ] +
                emb["am", ])/5
predict(model.pt, descartes, type = "nearest", top_n = 10)
```

```
##            term similarity rank
## 1             i  0.8336274    1
## 2         think  0.7580560    2
## 3        myself  0.7498994    3
## 4            we  0.7494881    4
## 5        really  0.7459691    5
## 6   [criticism]  0.7449573    6
```

Word embeddings

Representing documents

The package has a function called `doc2vec` to do this automatically. This function includes an additional scaling factor (see documentation) so the results are slightly different.

```
descartes <- doc2vec(model.pt, "i think therefore i am")
predict(model.pt, descartes, type = "nearest", top_n = 10)
```

```
## [[1]]
##              term similarity rank
## 1               i  0.9500304    1
## 2           think  0.8639066    2
## 3          myself  0.8546111    3
## 4              we  0.8541421    4
## 5          really  0.8501323    5
## 6     [criticism]  0.8489791    6
## 7              am  0.8417951    7
## 8       obviously  0.8406690    8
## 9              so  0.8392198    9
## 10           [am]  0.8369783   10
```

Word embeddings

Visualizing high-dimensional embeddings in low-dimensional space

- There are various algorithms available for visualizing word-embeddings in low-dimensional space
  - `PCA`, `t-SNE`, `UMAP`
- There are also browser-based interactive embedding explorers
  - See this example on the Tensorflow website

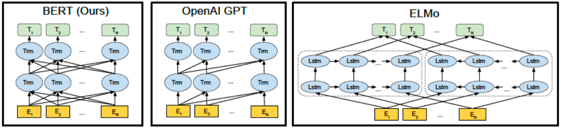
Contextualized embeddings

Limitations of existing approaches

- Word2vec and other embedding methods run into issues when dealing with *polysemy*
  - e.g. The vector for "crane" will be learned by averaging across different uses of the term
    - A bird
    - A type of construction equipment
    - Moving one's neck
  - "She had to crane her neck to see the crane perched on top of the crane".
- New methods have been developed to allow the vector for "crane" to vary according to different contexts
- The intuition here is that we want to take more context into account when constructing vectors

Contextualized embeddings

Architectures



Source: Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. “BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding.” In Proceedings of NAACL-HLT 2019, 4171–86. ACL.

Contextualized embeddings

Methodological innovations

- More complex, deeper neural networks
  - *Attention* mechanisms, *LSTM* architecture, *bidirectional transformers*
- Optimization over multiple tasks (not just a simple prediction problem like Word2vec)
- Character-level tokenization and embeddings
- Much more data and enormous compute power required
  - e.g. BERT trained on a 3.3 billion word corpus over 40 epochs, taking over 4 days to train on 64 TPU chips (each chip costs ~$10k).

Contextualized embeddings

On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? 🦜

Emily M. Bender*
ebender@uw.edu
University of Washington
Seattle, WA, USA

Timnit Gebru*
timnit@blackinai.org
Black in AI
Palo Alto, CA, USA

Angelina McMillan-Major
aymm@uw.edu
University of Washington
Seattle, WA, USA

Shmargaret Shmitchell
shmargaret.shmitchell@gmail.com
The Aether

ABSTRACT

The past 3 years of work in NLP have been characterized by the development and deployment of ever larger language models, especially for English. BERT, its variants, GPT-2/3, and others, most recently Switch-C, have pushed the boundaries of the possible both through architectural innovations and through sheer size. Using these pretrained models and the methodology of fine-tuning them for specific tasks, researchers have extended the state of the art on a wide array of tasks as measured by leaderboards on specific

alone, we have seen the emergence of BERT and its variants [39, 70, 74, 113, 146], GPT-2 [106], T-NLG [112], GPT-3 [25], and most recently Switch-C [43], with institutions seemingly competing to produce ever larger LMs. While investigating properties of LMs and how they change with size holds scientific interest, and large LMs have shown improvements on various tasks (§2), we ask whether enough thought has been put into the potential risks associated with developing them and strategies to mitigate these risks.

We first consider environmental risks. Echoing a line of recent

Bender, Emily M, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In Conference on Fairness, Accountability, and Transparency (*FAccT '21*).

Contextualized embeddings

Fine-tuning

- One of the major advantages of BERT and other approaches is the ability to "fine-tune" a model
  - We can train the model to accomplish a new task or learn the intricacies of a new corpus without retraining the mode
    - Although this can still take time and require quite a lot of compute power
- This means we could take an off-the-shelf, pre-trained BERT model and fine-tune it to an existing corpus
  - See this notebook for a Python example of fine-tuning BERT

Contextualized embeddings

Using contextualized embeddings in R

- Most contextualized embeddings require specialized programming languages optimized for large matrix computations like `PyTorch` and `Tensorflow`
- Once installed, I recommend using `keras`, a high-level package that can be used to implement various neural network methods without directly writing `Tensorflow` code.
- It is possible to work with these models in R, but you might be better off learning Python!

Summary

- Limitations of sparse representations of text
  - LSA allows us to project sparse matrix into a dense, low-dimensional representation
- Probabilistic language models allow us to directly model language use
- Word embeddings use a neural language model to better represent texts as dense vectors
  - Distributional semantics
  - Analogical reasoning
  - Sociological analysis of meaning and representations
- Recent methodological advances better incorporate context
  - Better semantic representations but huge financial and environmental costs