Word and Document Embeddings

Max Callaghan

2023-10-30

Objectives

Objectives

By now we have figured out how to **represent** texts using the “bag of words” model.

Today we will look at **word embeddings**, which are a basic building block for understanding much of much modern NLP.

Word embeddings

Why do we need word embeddings?

With the bag of words model, there are as many columns as there are unique words, and each column is completely independent.

Consider the following two texts

`"The acclaimed author penned novels based on her life"`

`"Nobel prize-winning writer writes autobiographical fiction"`

Are they similar? Would an bag of words model consider these similar texts?

Word similarity

Words themselves can be similar. A richer model will represent this similarity.

If this works, then the sentence

The **cat** sat on the mat.

will be much more similar to the sentence

The **dog** sat on the mat.

than it will be to the sentence

The **carbon** sat on the mat.

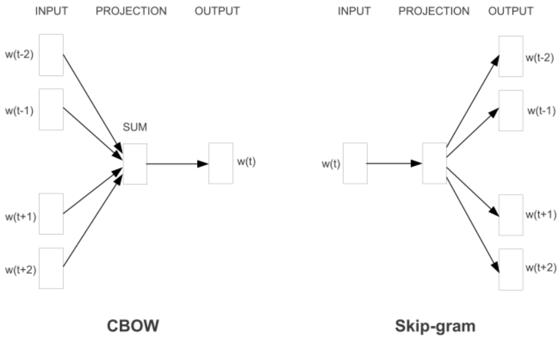
How can we find similar words

The intuition is that similar words appear in similar contexts

Word embeddings

Mikolov et al, 2013 developed a very efficient way to learn word embeddings for very large vocabularies.

Essentially, neural networks are trained on huge corpora to predict words from context (Continuous Bag of Words - CBOW) or context from words (Skip-gram)



from the sentence

`the cat sat on the mat`

we would have the following context, target pairs to predict

`([cat sat], the), ([the, sat, on], cat), ([the, cat, on, the], sat), ([cat, sat, the, mat], on), ([sat, on, mat],`

What do we learn?

The outcome of this learning is a vector representation of each word in `n` dimensions.

For example, if we learnt 3-dimensional embeddings on a corpus of documents, we might represent `cat` as `[0.3, 0.4, 0.6]`, `dog` as `[0.3, 0.42, 0.58]`, and `carbon` as `[0.5, -0.8, -0.1]`.

How would we calculate the similarity of these?

Glove Embeddings

Glove embeddings are trained on large Corpora (wikipedia, twitter, common crawl (loads of web data)).

We can load these pre-trained embeddings. Let's have a look at some of our words.

```
library(textdata)
library(coop)

N_DIM <- 100
glove <- embedding_glove6b(dimensions=N_DIM, dir="../embeddings")
word_matrix <- as.matrix(glove[,-1])
rownames(word_matrix) <- glove$token
words <- word_matrix[c("dog","cat","carbon"),]

words[,1:5]
```

```
##              d1      d2      d3       d4       d5
## dog     0.30817 0.30938 0.52803 -0.92543 -0.73671
## cat     0.23088 0.28283 0.63180 -0.59411 -0.58599
## carbon -0.86024 0.41537 0.77613 -0.14248  0.41046
```

Have a look at some words for yourself and identify their similarity in the embedding space. Do they represent meaningful distances?

Similarity with Glove Embeddings

Since each word is simply a vector, we can easily compute a similarity matrix of the words.

```
words <- word_matrix[c("dog","cat","carbon"),]
sims <- cosine(t(words))
sims
```

```
##              dog        cat     carbon
## dog    1.0000000 0.87980750 0.09022930
## cat    0.8798075 1.00000000 0.05027368
## carbon 0.0902293 0.05027368 1.00000000
```

Have a look at some words for yourself and identify their similarity in the embedding space. Do they represent meaningful distances?

Glove Embeddings in Python

```
import pandas as pd
import csv
from zipfile import ZipFile
N_DIMS = 100
z = ZipFile("../embeddings/glove6b/glove.6B.zip")
f = z.open(f'glove.6B.{N_DIMS}d.txt')
word_matrix = pd.read_table(
    f, sep=" ", index_col=0,
    header=None, quoting=csv.QUOTE_NONE
)
from sklearn.metrics.pairwise import cosine_similarity
word_list = ["dog","cat","carbon"]
sims = pd.DataFrame(cosine_similarity(word_matrix.loc[word_list]))
sims.index, sims.columns = word_list, word_list
sims
```

```
##              dog       cat    carbon
## dog     1.000000  0.879808  0.090229
## cat     0.879808  1.000000  0.050274
## carbon  0.090229  0.050274  1.000000
```

Comparing two word vectors

We can compare two vectors using the `cosine()` function from the https://cran.r-project.org/web/packages/coop/index.html library.

```
library(coop)
vec_a <- word_matrix["paris",]
vec_b <- word_matrix["france",]
cosine(vec_a, vec_b)
```

```
## [1] 0.7481587
```

Comparing two word vectors in python

Similarly, we just calculate 1 - the cosine distance from scipy.spatial.distance

```
from scipy.spatial.distance import cosine
vec_a = word_matrix.loc["paris"]
vec_b = word_matrix.loc["france"]
1 - cosine(vec_a, vec_b)
```

```
## 0.7481586531248817
```

Unrelated words

In pairs, one person starts by suggesting any word. The other pair member should come up with as dissimilar a word as possible. Continue with a word dissimilar to that. Note down your list of words and their similarity.

Finding similar words

If we want to find words that are similar to other words, then we can compare the vector for our target word with the vector for each other word.

```
vec_a <- word_matrix["cat",]
sims <- apply(word_matrix, 1, function(x) cosine(x,vec_a))
sims %>% sort(decreasing=T) %>% head()
```

```
##       cat       dog    rabbit      cats    monkey       pet
## 1.0000000 0.8798075 0.7424427 0.7323004 0.7288710 0.7190140
```

Conversely, we can also find dissimilar words, which can have a similarity < 0

```
sims %>% sort(decreasing=F) %>% head()
```

```
##     theros      vulso      lyssy  abbington  suhartono     chamni
## -0.5288887 -0.5141478 -0.5046386 -0.5018671 -0.5014641 -0.5009102
```

A function to find any word's neighbours

If we want to do this for a few different words, we might decide we don't want to write this code out every time. Let's build a function to do it for us

```
similar_words <- function(word, word_matrix) {
  vec_word <- word_matrix[word,]
  sims <- apply(word_matrix, 1, function(x) cosine(x,vec_word))
  return(sort(sims, decreasing=T))
}

similar_words("carbon", word_matrix) %>% head()
```

```
##     carbon    dioxide  emissions        co2 greenhouse      gases
##  1.0000000  0.8958776  0.8374466  0.8217504  0.8071149  0.8017597
```

A python function to find any word's neighbours

If we want to do this for a few different words, we might decide we don't want to write this code out every time. Let's build a function to do it for us

```
# function for similar words to x
def similar_words(word, word_matrix):
    vec_a = word_matrix.loc[word]
    sims = 1 - word_matrix.apply(cosine, axis=1, args=(vec_a,))
    return sims.sort_values(ascending=False)

similar_words("carbon",word_matrix).head(6)
```

```
## 0
## carbon        1.000000
## dioxide       0.895878
## emissions     0.837447
## co2           0.821750
## greenhouse    0.807115
## gases         0.801760
## dtype: float64
```

Encoded Linguistic Regularities and Patterns

Embedding spaces encode interesting regularities and patterns. If we subtract vector B from vector A, we get a vector that represents the *relationship* of those vectors.

If we subtract this vector from *another* vector C, we apply the same transformation, and get a vector D which is to C what B is to A. We can then find words which are close to D.

```
diff <- word_matrix["paris",]  - word_matrix["france",]
vec_d <- word_matrix["berlin",] - diff
sims <- apply(word_matrix, 1, function(x) cosine(x,vec_d))
sims %>% sort(decreasing=T) %>% head()
```

```
##   germany   austria   denmark    poland    berlin    france
## 0.8927663 0.7621856 0.7481993 0.7455099 0.7220174 0.7211105
```

Try out some other analogies !

Analogies in Python

```
diff = word_matrix.loc["paris"] - word_matrix.loc["france"]
vec_d = word_matrix.loc["berlin"] - diff
sims = 1 - word_matrix.apply(cosine, axis=1, args=(vec_d,))
sims.sort_values(ascending=False).head(6)
```

```
## 0
## germany    0.892766
## austria    0.762186
## denmark    0.748199
## poland     0.745510
## berlin     0.722017
## france     0.721111
## dtype: float64
```

Bias

Not all linguistic regularities and patterns are desirable.

```
##                john      jane    doctor     nurse
## john   1.0000000 0.5861599 0.3933398 0.2678903
## jane   0.5861599 1.0000000 0.3485967 0.4004849
## doctor 0.3933398 0.3485967 1.0000000 0.7521509
## nurse  0.2678903 0.4004849 0.7521509 1.0000000
```

Pre-trained word vectors encode historic and present biases (in particular racism and sexism) in how humans have used languages.

How this affects us depends on our application, but we should be particularly cautious when the application has the potential to *amplify* biases.

Start from stochastic parrots to explore more on bias, risk, and harms in NLP.

Polysemy

What words would be good neighbours for the word “flies”?

Polysemy

What words would be good neighbours for the word “flies”?

soars, flew, plane; or spider, bug, insect?

Polysemy

What words would be good neighbours for the word “flies”?

soars, flew, plane; or spider, bug, insect?

Each token has only one position in the embedding space regardless of how many senses it has

Learning our own embeddings

Word embeddings available online encode information about how words are used in the context of the training dataset.

We may care about how words are used in a *specific* context, in which case it may make sense to learn our own embeddings.

This usually makes sense when you have large numbers of documents. Check out text2vec for how to do this.

We may even care about how word use differs between subgroups (i.e. what terms are close to immigration in the Republican vs. Democrat space?). Check out Arthur Spirling's work on this.

Document Embeddings

Documents

Lets work out how to use embeddings to represent documents, and see if they are any good. We'll use our example documents from before, and put these into a document-feature matrix, and work out their similarity.

```
docs <- c(
  "The acclaimed author penned novels based on her life",
  "Nobel prize-winning writer writes autobiographical fiction"
)
dfmat <- docs %>% tokens(remove_punc=TRUE) %>%
  tokens_remove(pattern=stopwords("en")) %>%
  dfm()

print(cosine(as.vector(dfmat[1,]), as.vector(dfmat[2,])))
```

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

```
dfmat
```

```
## Document-feature matrix of: 2 documents, 12 features (50.00% sparse) and 0 docvars.
##        features
## docs     acclaimed author penned novels based life nobel prize-winning writer
##   text1          1      1      1      1     1    1     0             0      0
##   text2          0      0      0      0     0    0     1             1      1
##        features
## docs     writes
##   text1       0
```

Common features

Let's now find the words that are used in the document feature matrix, and the words that are in our embedding vocabulary, and select the parts of each matrix that uses these words. For simplicity first of all, we will select only the first 5 dimensions of the embedding space, and 4 features. We will also round the vectors to 1 decimal place

```
common_features <- intersect(colnames(dfmat),rownames(word_matrix))
common_features <- c("author","novels","writer","writes")

glove_dfmat <- dfmat[,common_features]
print(glove_dfmat)
```

```
## Document-feature matrix of: 2 documents, 4 features (50.00% sparse) and 0 docvars.
##        features
## docs     author novels writer writes
##   text1       1      1      0      0
##   text2       0      0      1      1
```

```
corpus_word_matrix <- round(word_matrix[common_features,1:5],1)
print(corpus_word_matrix)
```

```
##          d1   d2   d3   d4  d5
## author -0.4  0.2  0.0 -0.1 0.9
## novels -0.3  0.0 -0.2  0.1 1.1
## writer -0.7 -0.1 -0.2 -0.6 0.5
## writes -1.3  0.3  0.1 -0.6 0.9
```

Summing

We want a score in each dimension, for each document.

We can achieve this by multiplying the two matrices: $AB = C$.

This means that each cell $C_{i,j}$ is the dot product of the ith row of A and the jth column of B
$A_{i,} \cdot B_{,j}$

```
doc_matrix <- glove_dfmat %*% corpus_word_matrix
doc_matrix
```

```
## 2 x 5 Matrix of class "dgeMatrix"
##         d1  d2   d3   d4  d5
## text1 -0.7 0.2 -0.2  0.0 2.0
## text2 -2.0 0.2 -0.1 -1.2 1.4
```

Unsimplifying

Now we'll repeat this process but with the full feature set and the full set of dimensions

```
common_features <- intersect(colnames(dfmat),rownames(word_matrix))

glove_dfmat <- dfmat[,common_features]
corpus_word_matrix <- word_matrix[common_features,]
doc_matrix <- glove_dfmat %*% corpus_word_matrix
print(cosine(doc_matrix[1,], doc_matrix[2,]))
```

```
## [1] 0.8526549
```

The cosine similarity of these documents in the embedding space is very high, even though they share no words in common!

Embedding documents in python

Embedding documents in python works exactly the same

```
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np
docs = [
  "The acclaimed author penned novels based on her life",
  "Nobel prize-winning writer writes autobiographical fiction"
]
vec = CountVectorizer()
dfmat = vec.fit_transform(docs).todense()

common_features = set(word_matrix.index) & set(vec.get_feature_names_out())
common_features = list(common_features)
vocab_ids = [vec.vocabulary_[x] for x in common_features]
doc_matrix = dfmat[:,vocab_ids].dot(word_matrix.loc[common_features,])
1 - cosine(doc_matrix[0,].A1, doc_matrix[1,].A1)
```

```
## 0.7854257726317803
```

The cosine similarity of these documents in the embedding space is very high, even though they share no words in common!

Embedded manifestos

Now let's try embedding our manifesto sentences and reducing the dimensionality.

Can you write your own manifesto sentence and find the real sentence that is most similar to it?

Wrapup and Outlook

Wrapup

- Word embeddings form our first encounter with *fancy* NLP.

Wrapup

- Word embeddings form our first encounter with *fancy* NLP.
- We can place words, *and documents*, in a multidimensional embedding space.

Wrapup

- Word embeddings form our first encounter with *fancy* NLP.
- We can place words, *and documents*, in a multidimensional embedding space.
- This space *encodes* **abstract** but **meaningful** information about language

Wrapup

- Word embeddings form our first encounter with *fancy* NLP.
- We can place words, *and documents*, in a multidimensional embedding space.
- This space *encodes* **abstract** but **meaningful** information about language
- Encoding texts in this space allows us to do a **better job** at *some* tasks

Outlook

- Next week, we will go into *topic modelling*, on which there will be another assignment. Come to the class prepared by doing the reading

Outlook

- Next week, we will go into *topic modelling*, on which there will be another assignment. Come to the class prepared by doing the reading
- Please fill out this informal midterm evaluation link