CIS 8398
Advanced AI Topics in Business

#Working with Text Data

Yu-Kai Lin

Agenda

- One-hot encoding
- Word embedding

[**Acknowledgments**] The materials in the following slides are based on the source(s) below:

- Deep Learning with R by Francois Chollet and J.J. Allaire
- R interface to Keras

Prerequisites

```r
#install.packages("tidyverse")
#install.packages("keras")
install.packages("hashFunction")

library(tidyverse)
library(keras)
#install_keras() # will take a while; you should have done this already
```

Some of the deep neural networks are too large to train on a normal laptop, so you may use the ones that I have already trained:

```r
model <- load_model_hdf5("some_model_filename.h5")
```

Download pretrained models (and their histories) here:
https://www.dropbox.com/s/hpdhvzrx9p9etnl/deeplearning-pretrained.zip?dl=0

If you are using the VM, the pretrained model and history files are stored in C:/CIS8398/data/deeplearning-pretrained/

IMDB Movie Reviews Dataset

Download from here or here and extract the data to `<PROJECT_DIR>/data/aclImdb`.

Under **aclImdb** folder, there are two subfolders: **test** and **train**. Within each of these folders, there are a **neg** folder for negative reviews and a **pos** folder for positive reviews.

The filename of each txt file shows the review ID and the rating. For example, the file `aclImdb/test/neg/0_2.txt` has a review ID of 0 and a rating of 2.

The review ID ranges from 0 to 12499 and the rating ranges from 1 to 10.

Source: Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).

Working with text data

Deep-learning models don't take raw text as input: they only work with numeric tensors.

Vectorizing text is the process of transforming text into numeric tensors. This can be done in multiple ways:

- Segment text into words, and transform each word into a vector.
- Segment text into characters, and transform each character into a vector.
- Extract *n-grams* of words or characters, and transform each n-gram into a vector.

Collectively, the different units into which we can break down text (words, characters, or n-grams) are called *tokens*, and breaking text into such tokens is called *tokenization*.

Text to tokens to vectors

All text-vectorization processes consist of applying some tokenization scheme and then **associating numeric vectors with the generated tokens**.

There are multiple ways to associate a token with a vector. In this lecture, we'll learn two major ones:

- **one-hot encoding** of tokens
- **token embedding** (typically used exclusively for words, and called **word embedding**).

One-hot encoding

One-hot encoding is the most common, and most basic, way to turn a token into a vector.

A token is a word.

It consists of associating a unique integer index with every token and then turning this integer index i into a binary vector of size N (the size of the vocabulary); the vector is all zeros except for the ith entry, which is 1.

Word-level one-hot encoding (toy example)

```r
samples <- c("The cat sat on the mat.", "The dog ate my homework.")
token_index <- list()

for (sample in samples) {
  for (word in strsplit(sample, " ")[[1]]) {
    if (!word %in% names(token_index)) {
      # We don't attribute index 1 to anything. Start from 2.
      token_index[[word]] <- length(token_index) + 2
    }
  }
}

max_length <- 10 # We'll only consider the first 10 words in each sample.
results <- array(0, dim = c(length(samples),
                            max_length,
                            max(as.integer(token_index))))
for (i in 1:length(samples)) {
  sample <- samples[[i]]
  words <- head(strsplit(sample, " ")[[1]], n = max_length)
  for (j in 1:length(words)) {
    index <- token_index[[words[[j]]]]
    results[[i, j, index]] <- 1
  }
}
```

Columns are token indexes and rows are tokens in the sample.

```r
results[1,,] # First sample: The cat sat on the mat.
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
##  [1,]    0    1    0    0    0    0    0    0    0     0     0
##  [2,]    0    0    1    0    0    0    0    0    0     0     0
##  [3,]    0    0    0    1    0    0    0    0    0     0     0
##  [4,]    0    0    0    0    1    0    0    0    0     0     0
##  [5,]    0    0    0    0    0    1    0    0    0     0     0
##  [6,]    0    0    0    0    0    0    1    0    0     0     0
##  [7,]    0    0    0    0    0    0    0    0    0     0     0
##  [8,]    0    0    0    0    0    0    0    0    0     0     0
##  [9,]    0    0    0    0    0    0    0    0    0     0     0
##  [ reached getOption("max.print") -- omitted 1 row ]
```

```r
results[2,,] # Second sample: The dog ate my homework.
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
##  [1,]    0    1    0    0    0    0    0    0    0     0     0
##  [2,]    0    0    0    0    0    0    0    1    0     0     0
##  [3,]    0    0    0    0    0    0    0    0    1     0     0
##  [4,]    0    0    0    0    0    0    0    0    0     1     0
##  [5,]    0    0    0    0    0    0    0    0    0     0     1
##  [6,]    0    0    0    0    0    0    0    0    0     0     0
##  [7,]    0    0    0    0    0    0    0    0    0     0     0
##  [8,]    0    0    0    0    0    0    0    0    0     0     0
##  [9,]    0    0    0    0    0    0    0    0    0     0     0
##  [ reached getOption("max.print") -- omitted 1 row ]
```

To represent a document using a vector:

```
rbind(
  colSums(results[1,,]), # The cat sat on the mat.
  colSums(results[2,,])  # The dog ate my homework.
)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
## [1,]    0    1    1    1    1    1    1    0    0     0     0
## [2,]    0    1    0    0    0    0    0    1    1     1     1
```

```
token_index
```

```
## $The
## [1] 2
##
## $cat
## [1] 3
##
## $sat
## [1] 4
##
## $on
## [1] 5
##
## $the
## [1] 6
##
## $mat.
## [1] 7
##
## $dog
## [1] 8
##
## $ate
## [1] 9
##
## $my
## [1] 10
##
## $homework.
## [1] 11
```

Using Keras for word-level one-hot encoding

```
library(keras)
samples <- c("The cat sat on the mat.",
             "The dog ate my homework.")

# Creates a tokenizer, configured to only take into
# account the 10 most common words
tokenizer <- text_tokenizer(num_words = 10) |>
  fit_text_tokenizer(samples) # Builds the word index

word_index <- tokenizer$word_index

# Turns strings into lists of integer indices
sequences <- texts_to_sequences(tokenizer, samples)
one_hot_results <- texts_to_matrix(
  tokenizer, samples, mode = "binary")

one_hot_results
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    1    1    1    1    1    0    0    0     0
## [2,]    0    1    0    0    0    0    1    1    1     1
```

```
word_index
```

```
## $the
## [1] 1
##
## $cat
## [1] 2
##
## $sat
## [1] 3
##
## $on
## [1] 4
##
## $mat
## [1] 5
##
## $dog
## [1] 6
##
## $ate
## [1] 7
##
## $my
## [1] 8
##
## $homework
## [1] 9
```

Word embeddings

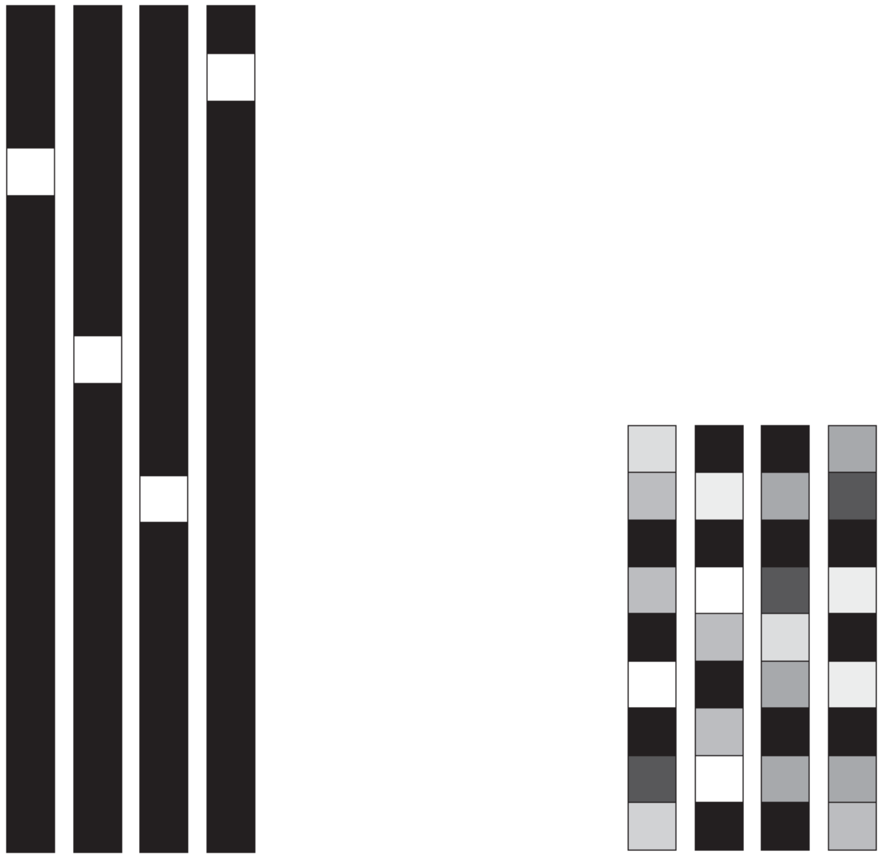
Whereas the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (same dimensionality as the number of words in the vocabulary), word embeddings are low-dimensional floating-point vectors (that is, dense vectors, as opposed to sparse vectors).

- One-hot encoding is hard-coded
- Word embeddings are **learned from data**

Word embeddings pack more information into far fewer dimensions.

- It's common to see word embeddings that are 256-dimensional, 512-dimensional, or 1,024-dimensional, when dealing with very large vocabularies.
- On the other hand, one-hot encoding words generally leads to vectors that are 20,000-dimensional or greater (capturing a vocabulary of 20,000 tokens, in this case).

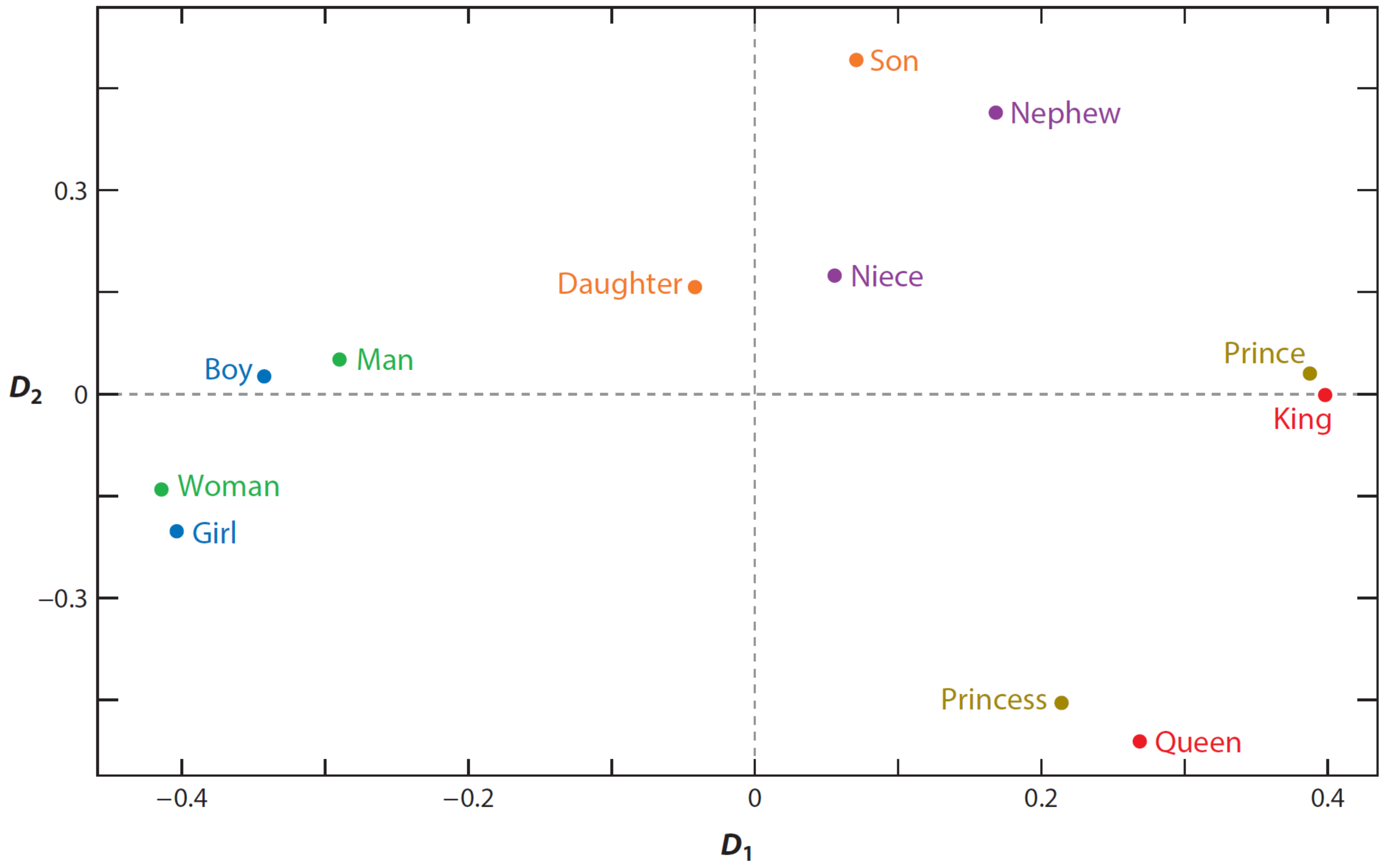
One-hot word vectors vs. Word embeddings



One-hot word vectors:
- Sparse
- High-dimensional
- Hardcoded

Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

2D projection of 300-dimension embeddings of male/female word pairs



Source: Stine, R. A. 2019. "Sentiment Analysis," Annual Review of Statistics and Its Application (6:1), pp. 287–308.

How to obotain word embeddings?

There are two ways to obtain word embeddings:

- Learn your own word embeddings from data. In this setup, we start with random word vectors and then learn word vectors in the same way we learn the weights of a neural network.
- Load into your model word embeddings that were precomputed using a different machine-learning task than the one we're trying to solve. These are called *pretrained word embeddings*.

Learn your own word embeddings

The geometric relationships between word vectors should reflect the semantic relationships between these words. Word embeddings are meant to map human language into a geometric space.

For instance, in a reasonable embedding space, we would expect synonyms to be embedded into similar word vectors; and in general, we would expect the geometric distance between any two word vectors to relate to the semantic distance between the associated words (words meaning different things are embedded at points far away from each other, whereas related words are closer).

Question!

How can we/machines know that two words (say, dog and canine) are synonyms?

This is the key puzzle regarding why/how word-embedding works.

Answer:

- **We learn how two words are related/synonyms based on the similarity of their surrounding words!**
- Word embedding leverages surrounding words to describe the focal word.

What makes a good word-embedding space depends heavily on your task.

The perfect word-embedding space for an English-language movie-review sentiment analysis model may look different from the perfect embedding space for an English language legal-document-classification model, because the importance of certain semantic relationships varies from task to task.

It's thus reasonable to learn a new embedding space with every new task. Fortunately, Keras makes this easy. It's about learning the weights of a layer using `layer_embedding`.

```
# Instantiating an embedding layer
embedding_layer <- layer_embedding(
  input_dim = 1000,  # the number of possible tokens
  output_dim = 64    # the dimensionality of the embeddings
)
```

`layer_embedding` is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors. It takes integers as input, it looks up these integers in an internal dictionary, and it returns the associated vectors. It's effectively a dictionary lookup:

Word index → Embedding layer → Corresponding word vector

Input to `layer_embedding`:

- A 2D tensor of integers, of shape (`samples`, `sequence_length`), where each entry is a sequence of integers.
- It can embed sequences of variable lengths: for instance, we could feed into the embedding layer with batches of shapes (32, 10) (batch of 32 sequences of length 10) or (64, 15) (batch of 64 sequences of length 15).
- All sequences in a batch must have the same length so sequences that are shorter than others should be padded with zeros, and sequences that are longer should be truncated.

Output from `layer_embedding`:

- A 3D floating-point tensor of shape (`samples`, `sequence_length`, `embedding_dimensionality`).
- Such a 3D tensor can then be processed by an RNN layer or a 1D convolutional layer (both will be introduced in the following sections).

Let's apply this idea to the IMDB movie-review sentiment-prediction task that we're already familiar with.

First, we'll quickly prepare the data. We'll restrict the movie reviews to the top **10,000 most common words** (as we did the first time we worked with this dataset) and cut off the reviews after **200 words**.

```
max_features <- 10000 # Number of words to consider as features
maxlen <- 200 # Cuts off the text after this number of words

imdb <- dataset_imdb(num_words = max_features)
c(c(x_train, y_train), c(x_test, y_test)) %<-% imdb # Loads the data

# Turns the lists of integers into a 2D tensor of shape (samples, maxlen)
# Will talk more about pad_sequences later
x_train <- pad_sequences(x_train, maxlen = maxlen)
x_test <- pad_sequences(x_test, maxlen = maxlen)
```

Next, we train a network for classification.

The network will learn **8-dimensional embeddings** for each of the **10,000 words**, turn the input integer sequences (2D integer tensor) into embedded sequences (3D float tensor), flatten the tensor to 2D, and train a single dense layer on top for classification.

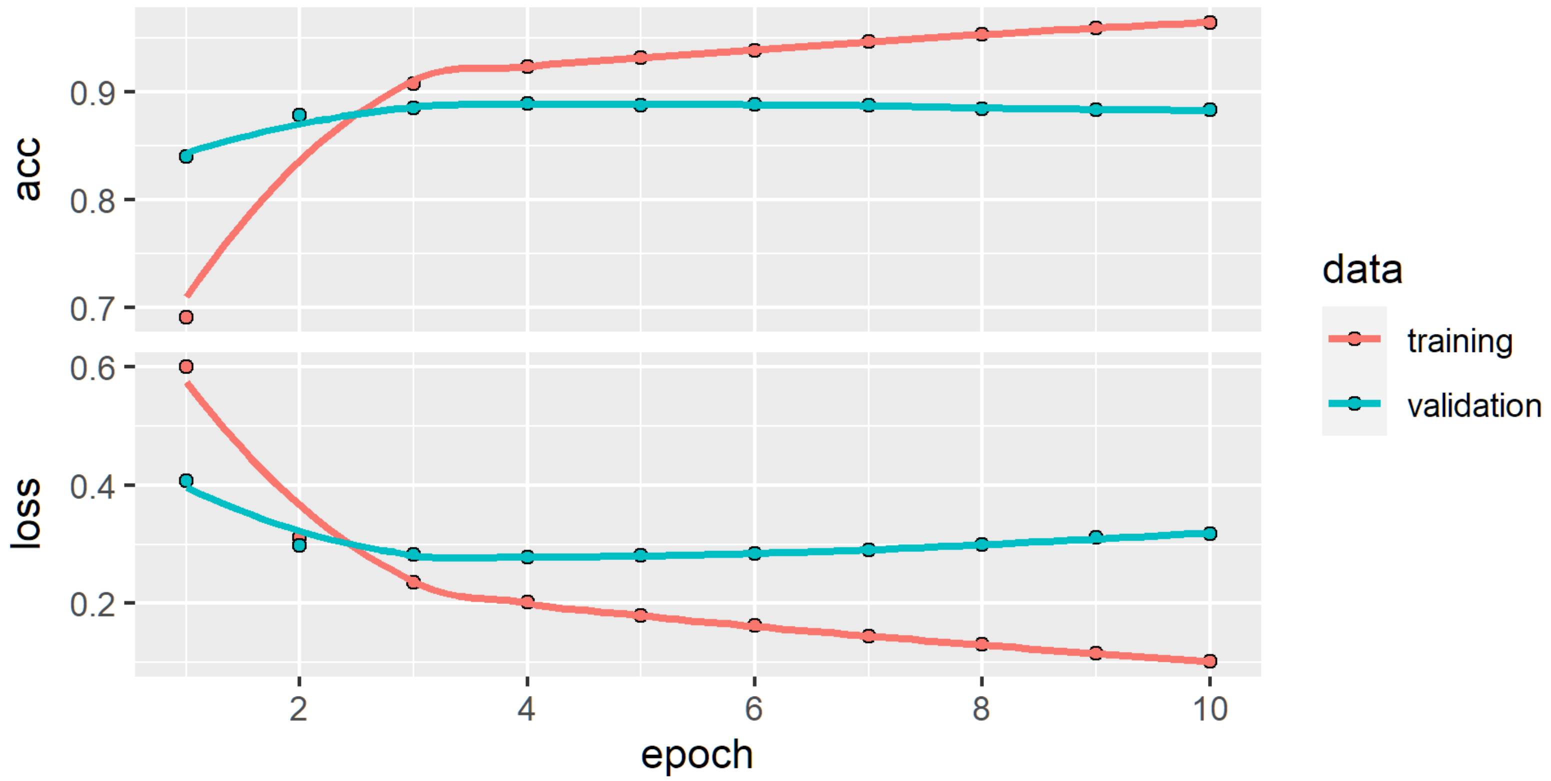
```
model <- keras_model_sequential() |>
  layer_embedding(input_dim = max_features, # num of unique words
                  input_length = maxlen, # num of words in a review
                  output_dim = 8) |>
  layer_flatten() |> # From (samples, maxlen, 8) to (samples, maxlen * 8)
  layer_dense(units = 1, activation = "sigmoid") # Adds the classifier on top

model |> compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

```
summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  embedding (Embedding)              (None, 200, 8)                  80000
##  flatten (Flatten)                  (None, 1600)                    0
##  dense (Dense)                      (None, 1)                       1601
## ================================================================================
## Total params: 81,601
## Trainable params: 81,601
## Non-trainable params: 0
## ________________________________________________________________________________
```

```r
history <- model |> fit(
  x_train, y_train,
  epochs = 10,
  batch_size = 32,
  validation_split = 0.2
)
```



To get the word embedding we just learned from data:

```
# retrieve the first layer (the embedding layer)
my_word_embedding = get_layer(model, index = 1) |>
  get_weights() %>% .[[1]]  # the base R pipe currently does not support .[[1]
str(my_word_embedding)
```

```
##  num [1:10000, 1:8] 0.0313 -0.1256 -0.0169 -0.0328 -0.0404 ...
```

```r
imdb_word_index =
  dataset_imdb_word_index()
imdb_word_index
```

```
## $fawn
## [1] 34701
##
## $tsukino
## [1] 52006
##
## $nunnery
## [1] 52007
##
## $sonja
## [1] 16816
##
## $vani
## [1] 63951
##
## $woods
## [1] 1408
##
## $spiders
## [1] 16115
##
## $hanging
## [1] 2345
##
```

```r
imdb_word_index_sub =
  imdb_word_index[map_lgl(
    imdb_word_index, ~.<=max_features)]
imdb_word_index_sub
```

```
## $woods
## [1] 1408
##
## $hanging
## [1] 2345
##
## $woody
## [1] 2289
##
## $arranged
## [1] 6746
##
## $bringing
## [1] 2338
##
## $wooden
## [1] 1636
##
## $errors
## [1] 4010
##
## $dialogs
## [1] 3230
```

To make predictions:

```
predictions = model |>
  predict(x_test) %>% # the base R pipe currently does not support `>`
  `>`(0.5) |> k_cast("int32") |>
  as.integer()
```

```
## 782/782 - 2s - 2s/epoch - 2ms/step
```

```
df = tibble(true_label = y_test, predicted_label = predictions)
df
```

```
## # A tibble: 25,000 × 2
##    true_label predicted_label
##         <int>           <int>
##  1          0               0
##  2          1               1
##  3          1               1
##  4          0               0
##  5          1               1
##  6          1               1
##  7          1               1
##  8          0               0
##  9          0               1
## 10          1               1
## # ℹ 24,990 more rows
```

Take aways:

We get to a validation accuracy of ~88%, which is pretty good.

But note that merely flattening the embedded sequences and training a single dense layer on top leads to a model that treats each word in the input sequence separately, without considering inter-word relationships and sentence structure.

It's much better to add recurrent layers or 1D convolutional layers on top of the embedded sequences to learn features that take into account each sequence as a whole. That's what we will be focusing on later in today's lecture.

Your turn

Follow the previous slides to create `my_word_embedding` and `imdb_word_index_sub`

- What are the embeddings for `horror` and `action`, respectively?
- What is the word that is most similar to `actor` based on the embeddings?
    - Hint: `euclidean_dist <- function(x, y) sqrt(sum((x - y)^2))`

Use pretrained word embeddings

Rationale: we often don't have enough data available to learn truly powerful features on our own (same rationale for why we were using pretrained convnets in image classification).

- Remember: A word embedding is generated based on surrounding words. So the more data you have, the more accurate/powerful your word embedding model will be.

Instead of learning word embeddings on your own, you can load and use pretrained embedding vectors.

There are various precomputed databases of word embeddings that we can download and use in a Keras embedding layer. **Word2vec** and **GloVe** are two famous ones.

Let's look at how we can get started using GloVe embeddings in a Keras model. The same method is valid for Word2vec embeddings database.

Processing the labels of the raw IMDB data:

```r
aclImdb_dir <- "data/aclImdb"
train_dir <- file.path(aclImdb_dir, "train")

labels <- c()
texts <- c()

for (label_type in c("neg", "pos")) {
  label <- switch(label_type, neg = 0, pos = 1)
  dir_name <- file.path(train_dir, label_type)
  for (fname in list.files(dir_name, pattern = glob2rx("*.txt"),
                           full.names = TRUE)) {
    texts <- c(texts, readChar(fname, file.info(fname)$size))
    labels <- c(labels, label)
  }
}
```

Tokenizing the text of the raw IMDB data:

```r
library(keras)
max_features <- 10000      # only consider the 10k most common words
maxlen <- 200              # only consider the first 200 words in each review
training_samples <- 200    # only use 200 reviews to train our model
validation_samples <- 5000
tokenizer <- text_tokenizer(num_words = max_features) |>
  fit_text_tokenizer(texts)
sequences <- texts_to_sequences(tokenizer, texts)
word_index = tokenizer$word_index
cat("Found", length(word_index), "unique tokens.\n")
```

```
## Found 88582 unique tokens.
```

```r
data <- pad_sequences(sequences, maxlen = maxlen)
labels <- as.array(labels)
cat("Shape of data tensor (Num Docs, Num Words in a Doc):", dim(data), "\n")
```

```
## Shape of data tensor (Num Docs, Num Words in a Doc): 25000 200
```

```r
cat('Shape of label tensor (Num Docs):', dim(labels), "\n")
```

```
## Shape of label tensor (Num Docs): 25000
```

Let's make sense of these different objects:

```
word_index
```

```
## $the
## [1] 1
##
## $and
## [1] 2
##
## $a
## [1] 3
##
## $of
## [1] 4
##
## $to
## [1] 5
##
## $is
## [1] 6
##
## $br
## [1] 7
##
## $`in`
## [1] 8
```

```
sequences
```

```
## [[1]]
##   [1]   62    4    3  129   34   44 7576 1414   15    3
##  [16]  633  133   12    6    3 1301  459    4 1751  209
##  [31]   80   32 2137 1110 3008   31    1  929    4   42
##  [46]    1  223   55   16   54  828 1318  847  228    9
##  [61]  145   36    1  996  141   27  676  122    1  411
##  [76]    5    3  837   20    3 1755  646   42  125   71
##  [91]   49  624   31  702   84  702  378 3493    2 8422
##  [ reached getOption("max.print") -- omitted 4 entries ]
##
## [[2]]
##   [1]  667 5399  295    1   88 3789 1086 9259    4   29
##  [16]    4  673    9    6  368 2306    1  226    6 1293
##  [31]  106    6    3 6192    4  761    2   21    3  160
##  [46]    3 1059    6 3537    8   32 1810 6538 1374    2
##  [61]  172 4892    8    1  271    6  318    8    3  693
##  [76]    7 1302  613    6   11  159  894 2666    4  744
##  [91]    3   17    6   11 3538    9  149 5229  744  408
##  [ reached getOption("max.print") -- omitted 19 entries
##
## [[3]]
##   [1]   11   19   66    3  173    4 2332    2    1  411
##  [16]  153  164    2  612  384   11   19  177    7    7
```

```r
texts[1] # first review
```

Story of a man who has unnatural feelings for a pig. Starts out with a opening scene that is a terrific example of absurd comedy. A formal orchestra audience is turned into an insane, violent mob by the crazy chantings of it's singers. Unfortunately it stays absurd the WHOLE time with no general narrative eventually making it just too off putting. Even those from the era should be turned off. The cryptic dialogue would make Shakespeare seem easy to a third grader. On a technical level it's better than you might think with some good cinematography by future great Vilmos Zsigmond. Future stars Sally Kirkland and Frederic Forrest can be seen briefly.

Story of a man who has unnatural feelings for a pig. Starts out with a opening scene that is a terrific example of absurd comedy. A formal orchestra audience is turned into an insane, violent mob by the crazy chantings of it's singers. Unfortunately it stays absurd the WHOLE time with no general narrative eventually making it just too off putting. Even those from the era should be turned off. The cryptic dialogue would make Shakespeare seem easy to a third grader. On a technical level it's better than you might think with some good cinematography by future great Vilmos Zsigmond. Future stars Sally Kirkland and Frederic Forrest can be seen briefly.

```r
data[1, 1:100] # first 100 tokens of the first review; where comes the 0's?
```

```
##   [1]   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##  [19]   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##  [37]   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##  [55]   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##  [73]   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##  [91]   0   0   0   0   0   0  62   4   3 129
```

```r
data[1, 101:200] # last 100 tokens of the first review
```

```
##   [1]   34   44 7576 1414   15    3 4252  514   43   16    3  633  133   12    6
##  [16]    3 1301  459    4 1751  209    3 7693  308    6  676   80   32 2137 1110
##  [31] 3008   31    1  929    4   42 5120  469    9 2665 1751    1  223   55   16
##  [46]   54  828 1318  847  228    9   40   96  122 1484   57  145   36    1  996
##  [61]  141   27  676  122    1  411   59   94 2278  303  772    5    3  837   20
##  [76]    3 1755  646   42  125   71   22  235  101   16   46   49  624   31  702
##  [91]   84  702  378 3493    2 8422   67   27  107 3348
```

Story of a man who has unnatural feelings for a pig. Starts out with a opening scene that is a terrific example of absurd comedy. A formal orchestra audience is turned into an insane, violent mob by the crazy chantings of it's singers. Unfortunately it stays absurd the WHOLE time with no general narrative eventually making it just too off putting. Even those from the era should be turned off. The cryptic dialogue would make Shakespeare seem easy to a third grader. On a technical level it's better than you might think with some good cinematography by future great Vilmos Zsigmond. Future stars Sally Kirkland and Frederic Forrest can be seen briefly.

```
map_chr(data[1,97:110], ~names(word_index)[.])
```

```
##  [1] "story"     "of"        "a"         "man"       "who"       "has"      
##  [7] "unnatural" "feelings"  "for"       "a"         "pig"       "starts"   
## [13] "out"       "with"
```

```
map_chr(data[1,190:200], ~names(word_index)[.])
```

```
##  [1] "future"  "great"   "future"  "stars"   "sally"   "and"     "forrest"
##  [8] "can"     "be"      "seen"    "briefly"
```

Your turn

Follow the previous slides to create `labels`, `texts`, `word_index`, `sequences`, and `data`

Show the label, text, word index, sequence, and data (pad_sequences) for the second review

Creating a training set and a validation set:

```r
# Splits the data into a training set and a validation set,
# but first shuffles the data, because we're starting with
# data in which samples are ordered (all negative first, then all positive)
set.seed(123)
indices <- sample(1:nrow(data))

training_indices <- indices[1:training_samples]
validation_indices <- indices[(training_samples + 1):
                              (training_samples + validation_samples)]

x_train <- data[training_indices,]
y_train <- labels[training_indices]
x_val <- data[validation_indices,]
y_val <- labels[validation_indices]
```

Download the GloVe Word Embeddings

Go to https://nlp.stanford.edu/projects/glove, and download the precomputed embeddings from 2014 English Wikipedia.

It's an **822 MB** zip file called glove.6B.zip, containing 4 embedding vectors files for 400,000 words.

- `glove.6B.50d.txt`: 50-dimensional embedding vectors
- `glove.6B.100d.txt`: 100-dimensional embedding vectors
- `glove.6B.200d.txt`: 200-dimensional embedding vectors
- `glove.6B.300d.txt`: 300-dimensional embedding vectors

Unzip `glove.6B.zip` to `<PROJECT_DIR>/data/glove.6B/`

Preprocessing the embeddings

Let's parse the unzipped file (a .txt file) to build an index that maps words (as strings) to their vector representation (as number vectors).

```r
glove_dir = "data/glove.6B/"

lines <- readLines(file.path(glove_dir, "glove.6B.50d.txt"))
embeddings_index <- new.env(hash = TRUE, parent = emptyenv())

for (i in 1:length(lines)) {
  line <- lines[[i]]
  values <- strsplit(line, " ")[[1]]
  word <- values[[1]]
  embeddings_index[[word]] <- as.double(values[-1])
}
cat("Found", length(embeddings_index), "word vectors.\n")
```

```
## Found 400000 word vectors.
```

Build an embedding matrix:

Next, we'll build an embedding matrix that we can load into an embedding layer. It must be a matrix of shape (`max_features`, `embedding_dim`), where each entry *i* contains the embedding_dim-dimensional vector for the word of index *i* in the reference word index (built during tokenization).

```
embedding_dim <- 50

embedding_matrix <- array(0, dim = c(max_features, embedding_dim)) # 10k x 50
for (word in names(word_index)) { # for every word
  index <- word_index[[word]]     # get its index
  if (index < max_features) {     # only consider the top 10k words
    # get the word's embedding vector from GloVe
    embedding_vector <- embeddings_index[[word]]
    if (!is.null(embedding_vector)) { # if GloVe has the embedding vector
      # index 1 isn't supposed to stand for any word or token
      # --it's a placeholder. So we skip 1 here:
      embedding_matrix[index+1,] <- embedding_vector
    }
  }
}
```

```
str(embedding_matrix)
```

```
##  num [1:10000, 1:50] 0 0.418 0.268 0.217 0.709 ...
```

```
embedding_matrix[1, ] # the first row is set to be a vector of zeros
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [39] 0 0 0 0 0 0 0 0 0 0 0 0
```

```
embedding_matrix[2, ] # the second row is the word with index 1
```

```
##  [1]  0.41800000  0.24968000 -0.41242000  0.12170000  0.34527000 -0.04445700
##  [7] -0.49688000 -0.17862000 -0.00066023 -0.65660000  0.27843000 -0.14767000
## [13] -0.55677000  0.14658000 -0.00950950  0.01165800  0.10204000 -0.12792000
## [19] -0.84430000 -0.12181000 -0.01680100 -0.33279000 -0.15520000 -0.23131000
## [25] -0.19181000 -1.88230000 -0.76746000  0.09905100 -0.42125000 -0.19526000
## [31]  4.00710000 -0.18594000 -0.52287000 -0.31681000  0.00059213  0.00744490
## [37]  0.17778000 -0.15897000  0.01204100 -0.05422300 -0.29871000 -0.15749000
## [43] -0.34758000 -0.04563700 -0.44251000  0.18785000  0.00278490 -0.18411000
## [49] -0.11514000 -0.78581000
```

```
                      # (the word  is 'the')
```

Your turn

You should have `word_index` in your R environment from the earlier Your Turn exercise.

Follow the previous slides to create `embedding_matrix`

- What are the embeddings for `horror` and `action`, respectively?
    - Hint 1: You can obtain the index of a specific word from `word_index`
    - Hint 2: Row **k** in `embedding_matrix` is the embedding vector for the word with the index **k-1**

- What is the word that is most similar to `actor` based on the embeddings?
    - Hint: `euclidean_dist <- function(x, y) sqrt(sum((x - y)^2))`

Model definition:

```
model_with_glove <- keras_model_sequential() |>
  layer_embedding(input_dim = max_features,
                  input_length = maxlen,
                  output_dim = embedding_dim) |>
  layer_flatten() |>
  layer_dense(units = 32, activation = "relu") |>
  layer_dense(units = 1, activation = "sigmoid")

summary(model_with_glove)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  embedding_1 (Embedding)            (None, 200, 50)                 500000
##  flatten_1 (Flatten)                (None, 10000)                   0
##  dense_2 (Dense)                    (None, 32)                      320032
##  dense_1 (Dense)                    (None, 1)                       33
## ================================================================================
## Total params: 820,065
## Trainable params: 820,065
## Non-trainable params: 0
## ________________________________________________________________________________
```

Load pretrained word embeddings into the embedding layer and train the rest of the layers:

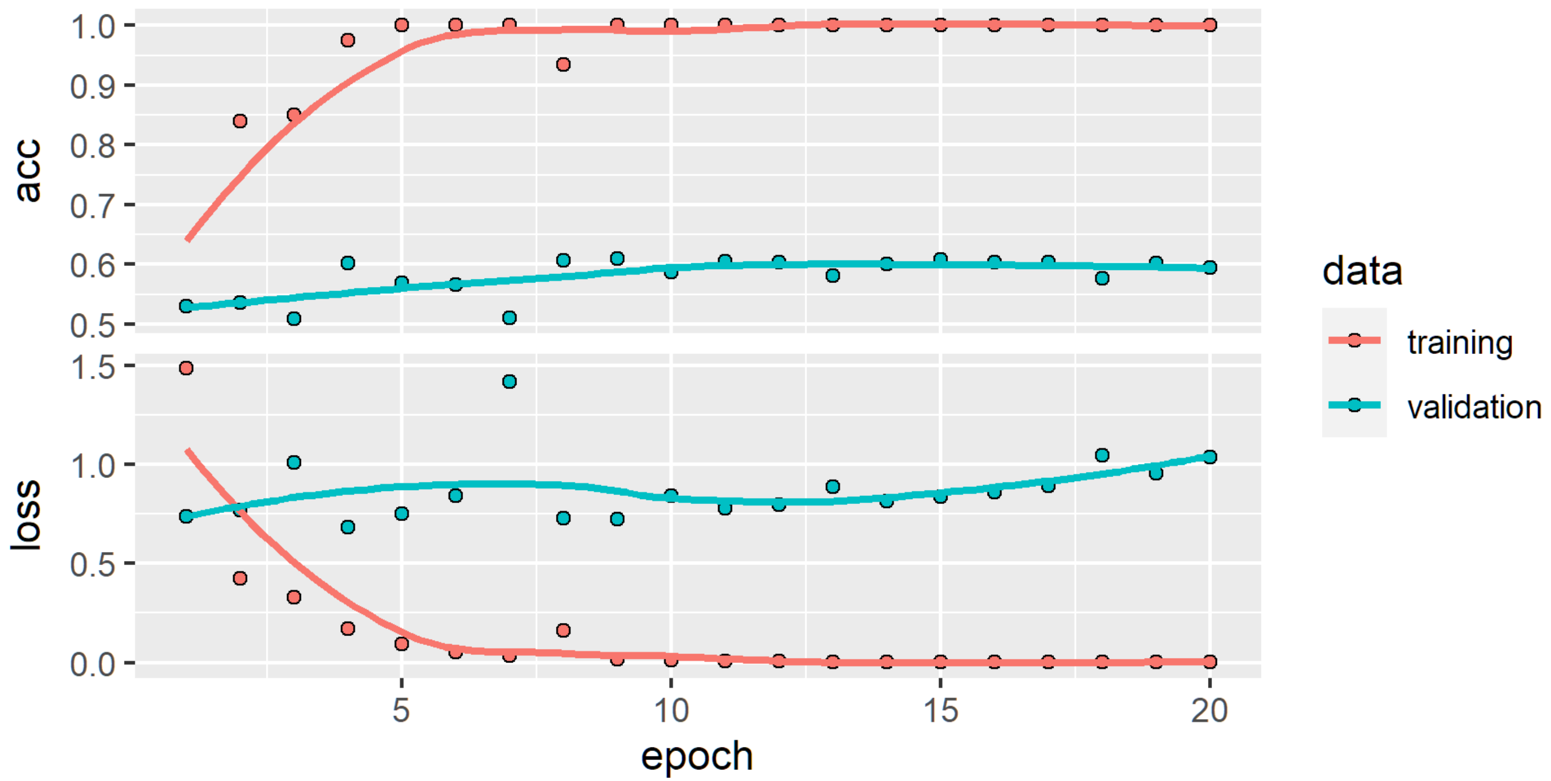
```r
get_layer(model_with_glove, index = 1) |>   # manually configure the embedding
  set_weights(list(embedding_matrix)) |>  # set the weights based on GloVe
  freeze_weights() # do not update the weights in this layer anymore

model_with_glove |> compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history <- model_with_glove |> fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 32,
  validation_data = list(x_val, y_val)
)

model_with_glove |> save_model_hdf5("imdb_word_embedding_with_glove.h5")
```

```
plot(history)
```



Note: The accuracy in validation set (~0.6) is much lower than what we saw in slide #25 because here we only use 200 reviews (`training_samples <- 200` on slide #33) to train the model (to mimic the situation when we do not have a large text corpus).

Let's compare that model with another one without pretrained word embeddings:

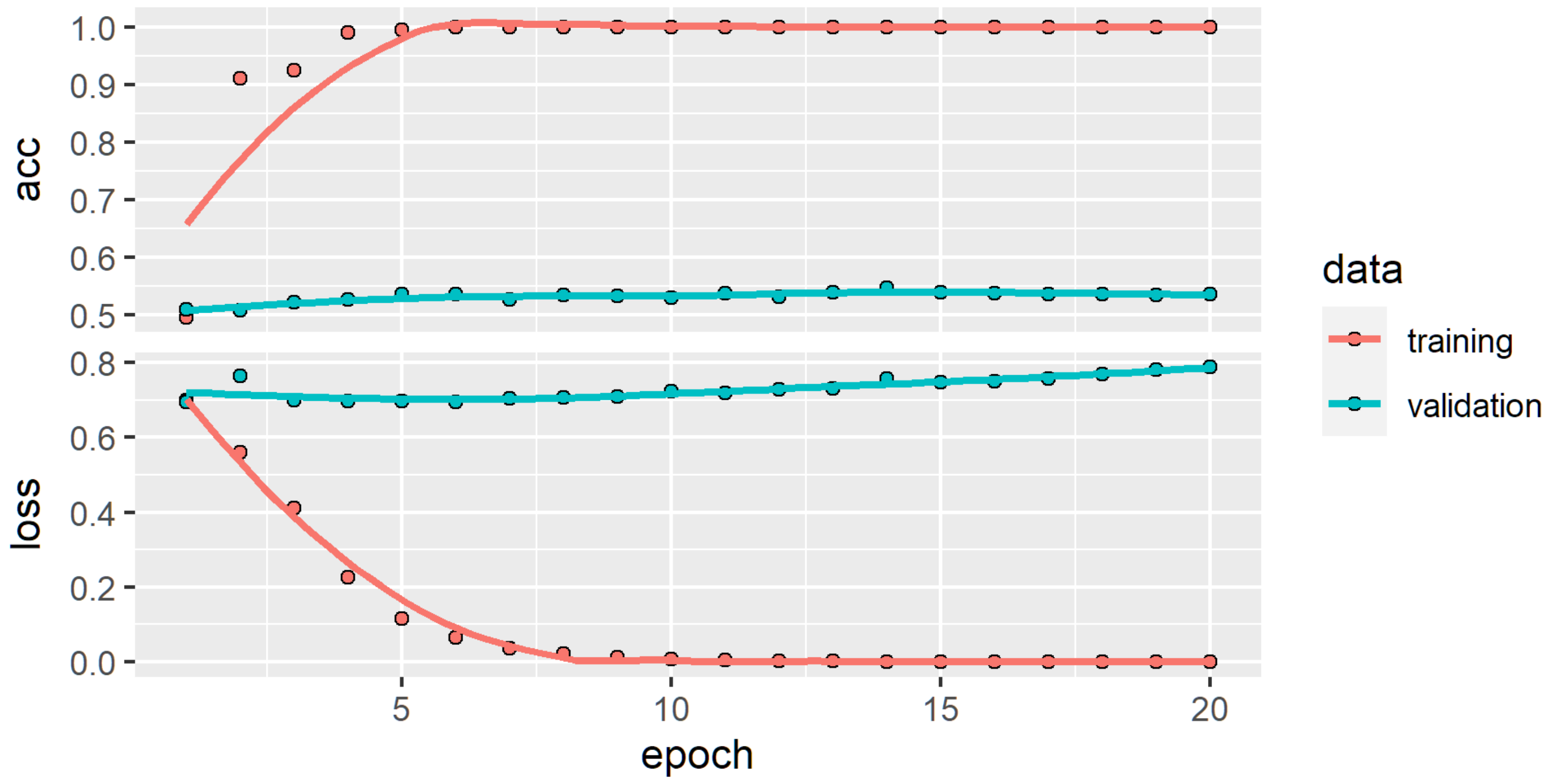
```r
model_without_glove <- keras_model_sequential() |>
  layer_embedding(input_dim = max_features,
                  input_length = maxlen,
                  output_dim = embedding_dim) |>
  layer_flatten() |>
  layer_dense(units = 32, activation = "relu") |>
  layer_dense(units = 1, activation = "sigmoid")

model_without_glove |> compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history2 <- model_without_glove |> fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 32,
  validation_data = list(x_val, y_val)
)

model_without_glove |> save_model_hdf5("imdb_word_embedding_without_glove.h5")
```

```
plot(history2)
```



The accuracy in validation set is around 0.55, which is lower than the previous one with GloVe (~0.6).

Finally, let's evaluate the model on the test data. First, we need to tokenize the test data.

```
test_dir <- file.path(aclImdb_dir, "test")
test_labels <- c()
test_texts <- c()

for (label_type in c("neg", "pos")) {
  label <- switch(label_type, neg = 0, pos = 1)
  dir_name <- file.path(test_dir, label_type)
  for (fname in list.files(dir_name, pattern = glob2rx("*.txt"),
                           full.names = TRUE)) {
    test_texts <- c(test_texts, readChar(fname, file.info(fname)$size))
    test_labels <- c(test_labels, label)
  }
}

sequences <- texts_to_sequences(tokenizer, test_texts)
x_test <- pad_sequences(sequences, maxlen = maxlen)
y_test <- as.array(test_labels)
```

```r
model_with_glove |>
  evaluate(x_test, y_test)
```

```
## 782/782 - 1s - loss: 0.8437 - accuracy: 0.5898 - 1s/epoch - 1ms/step

##      loss  accuracy
## 0.8437057 0.5898000
```

```r
model_without_glove |>
  evaluate(x_test, y_test)
```

```
## 782/782 - 1s - loss: 0.8060 - acc: 0.5202 - 939ms/epoch - 1ms/step

##      loss       acc
## 0.8060228 0.5202400
```

We can see that the pretrained word embedding can improve the accuracy.

Your turn

Try changing each of the following settings and see if the performance is different than what we saw in the previous slides:

- Use 20 words in each review instead of 200 (`maxlen`)
- Consider 5,000 words instead of 10,000 words (`max_features`)
- Use 2,000 reviews for training instead of 200 (`training_samples`)