Neural Networks: Text classifier

Previously we built a classification tree using bag-of-words features to predict whether a tweet made by Donald Trump was made before or after he became president.

Here we look at a series of notebooks that build different kinds of neural networks to do the same thing.

In this first notebook, we build a standard feedforward network (a multilayer perceptron) that uses bag-of-words features. The input data is the same as the classification tree we built previously; only the classification approach changes.

# Multilayer perceptron with bag-of-words features

library(readr)  
library(stringr)  
library(lubridate)  
library(tidytext)  
library(keras)  
use\_virtualenv("myenv", required = TRUE)   
  
load("processed\_trump\_tweets.Rdata")

The following objects are saved in processed\_trump\_tweets.Rdata:

* tweets\_sample (4000x11): data frame containing the actual text used in 4000 tweets (column text), response (column is\_potus), and additional variables. There are 2000 tweets made before, and 2000 tweets made during, Trump’s presidency.
* seq\_ok (4000x1): a logical variable indicating whether each tweet in tweets\_sample contains more than 5 tokens after tokenization. Tweets with seq\_id == TRUE are retained, else discarded.
* all\_ids (3438x1): contains an identifier (id\_str) for all retained tweets (all tweets where seq\_id == TRUE).
* bag\_of\_words (3438x203): a wide-format data frame containing, for all retained tweets (>5 tokens), bag-of-words features (frequency counts) for the 201 most commonly used words, plus columns for tweet id (id\_str) and response (is\_potus).
* training\_ids (2406x1): id\_str for all tweets that should be used for training.
* test\_ids (1032x1): id\_str for all tweets that should be used for testing
* test\_tweets\_bag (1032x202): format as for bag\_of\_words except the id\_str column has been removed. Contains tweets that should be considered part of the test dataset.
* test\_tweets\_txt (1032x10): data frame containing the actual text used in test tweets (column text), response (column is\_potus), and additional variables.
* train\_tweets\_bag (2406x202): format as for test\_tweets\_bag except these tweets should be considered part of the training dataset.
* train\_tweets\_txt (2406x10): as for text\_tweets\_txt, but for training tweets.

We restrict our models to longer tweets that have no fewer than 5 tokens (after tokenization described in the next section). This part isn’t needed for all the models we look at, but we want to treat all models the same for comparative purposes → apply this step to all models.

tweets\_sample\_valid <- tweets\_sample[seq\_ok, ]

We then split up the training and test set.

training\_rows <- which(tweets\_sample\_valid$id\_str %in% training\_ids$id\_str)  
  
# Converting bag-of-words features into numeric matrices  
train <- list()  
test <- list()  
train$x <- as.matrix(bag\_of\_words[training\_rows,-c(1,2)])  
test$x <- as.matrix(bag\_of\_words[-training\_rows,-c(1,2)])  
  
train$y <- as.integer(bag\_of\_words$is\_potus[training\_rows])  
test$y <- as.integer(bag\_of\_words$is\_potus[-training\_rows])

The neural network is created by stacking layers. Here this needs three main decisions, given below with the way we’re doing it in italics.

* How to represent the text? *bag-of-words features*
* How many layers to use in the model? *one dense layer below, but feel free to experiment*
* How many *hidden units* to use for each layer? *32 used below, but feel free to experiment*

The labels to predict are either 0 (tweet made before presidency) or 1 (tweet made during presidency).

model <- keras\_model\_sequential() %>%  
 layer\_dense(units = 32, input\_shape = c(201), activation = "relu") %>%  
 layer\_dense(units = 1, activation = "sigmoid")  
  
summary(model)

* First dense layer: 32 hidden units, ReLU activation, input shape = 201 features (bag-of-words size).
* Output layer: 1 unit with sigmoid activation for binary classification.

Compile the model:

model %>% compile(  
 loss = "binary\_crossentropy",  
 optimizer = "adam",  
 metrics = "accuracy")

* Specifies binary cross-entropy as the loss function for two-class classification.
* Uses the Adam optimizer for efficient training.
* Tracks accuracy as a performance metric.

Train the model:

history <- model %>% fit(train$x, train$y, epochs = 50, batch\_size = 128, verbose = 0)   
plot(history)

And evaluate it’s performance on the test set:

results <- model %>% evaluate(test$x, test$y, batch\_size=128, verbose = 2)

# Multilayer perceptron with word embeddings

* Builds a multilayer perceptron to predict if a Trump tweet was made before or after his presidency.
* Unlike the previous model, it uses the raw tweet text directly instead of pre-computed bag-of-words features.
* Introduces the concept of word embeddings, which represent words as dense numeric vectors to capture their meaning and relationships.

Here we use Keras to tokenize the tweets.

* Tokenization process: Keras breaks each tweet into individual words (tokens) and assigns each unique word a unique integer ID.
* The result is that every tweet is represented as a sequence of integers, where each integer corresponds to a specific word in the tokenizer’s vocabulary.
* This numeric representation allows the neural network to process textual data, since models cannot directly work with raw text.
* By default, the tokenizer splits text at the word level, but you can customize it to tokenize at other levels (e.g., characters or subwords) by adjusting its parameters.
* You can explore all available options and settings for tokenization using ?text\_tokenizer in R, such as setting vocabulary size limits or handling rare words.

max\_features <- 1000 # choose max\_features most popular words  
tokenizer = text\_tokenizer(num\_words = max\_features)   
fit\_text\_tokenizer(tokenizer, tweets\_sample$text)

* Creates a Keras tokenizer object that will map words to integer IDs based on their frequency.
* Fits the tokenizer to the tweet text data so it learns the vocabulary and assigns integer indices to each unique word.

tweets\_sample$text[1] # view the first tweet  
tts <- texts\_to\_sequences(tokenizer, tweets\_sample$text[1]) # tokenize the tweet

* Converts the first tweet into a sequence of integers, where each integer represents a word in the tokenizer’s vocabulary.
* The output is a list containing the sequence of integers for that tweet.

tokenizer$word\_index[tts[[1]]] # translates the integers back into their corresponding words.

sequences = tokenizer$texts\_to\_sequences(tweets\_sample$text) # tokanize all tweets

In later models we need to remove tweets with just a single word, or we get an error. So here we’ll do the same, just to keep the datasets comparable between models. I also throw out very short tweets (less than 5 words), but this is not strictly needed.

y <- as.integer(tweets\_sample$is\_potus[seq\_ok])  
sequences <- sequences[which(seq\_ok)]  
tweets\_sample\_valid <- tweets\_sample[seq\_ok, ]

We then split up the training and test set.

training\_rows <- which(tweets\_sample\_valid$id\_str %in% training\_ids$id\_str)  
  
train <- list()  
test <- list()  
train$x <- sequences[training\_rows]  
test$x <- sequences[-training\_rows]  
  
train$y <- y[training\_rows]  
test$y <- y[-training\_rows]

MLPs require inputs to be the same size – so sequences must be the same length. Currently this isn’t the case – sequences are of different lengths. So we “pad” the shorter sequences with zeros so that all padded sequences are the same length. We also need to set a maximum allowed sequence length, otherwise if there are just one or two much longer sentences, we end up with a lot of wasteful padding. Let’s see how long each of the tokenized tweets are:

hist(unlist(lapply(sequences, length)), main = "Sequence length after tokenization")

* Calculates the length of each tweet after tokenization.
* Plots a histogram to visualize the distribution of tweet lengths.
* Helps decide on an appropriate maximum sequence length for padding.

A maximum length around 30 seems reasonable. Now pad the sequences:

maxlen <- 32   
x\_train <- train$x %>% pad\_sequences(maxlen = maxlen)  
x\_test <- test$x %>% pad\_sequences(maxlen = maxlen)

* Neural networks require inputs of the same size, but tweets have different lengths.
* Uses pad\_sequences() to add zeros to the beginning of shorter tweets until each tweet has exactly maxlen = 32 words.
* Ensures consistent input dimensions for the model.

We can now define the model, using an embedding dimension of 10 (a vector of length 10 is used to represent each word).

model <- keras\_model\_sequential() %>%   
 layer\_embedding(max\_features, output\_dim = 10, input\_length = maxlen) %>%  
 layer\_dropout(0.2) %>%  
 layer\_flatten() %>%  
 layer\_dense(32, activation = "relu") %>%  
 layer\_dense(1, activation = "sigmoid")  
  
summary(model)

* Embedding layer:
  + Maps each integer (word) to a dense vector of length 10 (output\_dim = 10).
  + Learns meaningful word representations during training.
* Dropout layer: Prevents overfitting by randomly dropping 20% of units during training.
* Flatten layer: Converts the 2D embedding output into a single vector.
* Dense hidden layer: 32 neurons with ReLU activation for non-linearity.
* Output layer: 1 neuron with sigmoid activation for binary classification (before vs during presidency).
* Neural networks need numbers:
  + Neural networks perform mathematical operations like addition, multiplication, and matrix transformations.
  + Because of this, their inputs must be numeric values, not text, images, or other raw data types.
* Challenge with text data:
  + Words are discrete symbols, not numbers, so they cannot be processed directly by a neural network.
  + E.g. the words “president” and “tweet” have no inherent numeric meaning.
* Solution – numeric vector representations:
  + Each unique word (or other categorical entity like a user or movie) is assigned a numeric vector.
  + These vectors are learnable parameters that the model adjusts during training to capture useful patterns.
  + Example:
    - “president” [0.25, -0.18, 0.43, …]
    - “tweet” [0.09, 0.52, -0.33, …]
* What embeddings are:
  + An embedding is a function that maps a discrete unit (e.g., a word) to a dense, high-dimensional vector.
  + These vectors capture semantic relationships between words, so similar words have similar representations.
  + E.g. vectors for “king” and “queen” may be close together in the embedding space.
* How embeddings work internally:
  + The embedding layer can be thought of as a lookup table, implemented as a matrix:
    - Rows = unique words in the vocabulary.
    - Columns = vector dimensions (features for each word).
  + When a word appears in a tweet, its corresponding row vector is retrieved and passed into the network.
* Why this is powerful:
  + Instead of manually defining meaning, the neural network learns embeddings automatically based on the task.
  + This allows the model to detect patterns such as sentiment, context, or stylistic differences between tweets.
  + The same concept applies beyond words — embeddings are also used for users, products, movies, etc., in recommendation systems.

We can now compile the model:

model %>% compile(  
 loss = "binary\_crossentropy",  
 optimizer = "adam",  
 metrics = "accuracy")

* Loss function: Binary cross-entropy, ideal for binary classification tasks.
* Optimizer: Adam, for efficient and adaptive learning.
* Tracks accuracy during training and evaluation.

Train the model:

history <- model %>%  
 fit(x\_train, train$y,batch\_size = 64, epochs = 10, verbose = 0)  
plot(history) # visualize loss and accuracy over epochs

And evaluate the model on the test set:

results <- model %>% evaluate(x\_test, test$y, batch\_size = 64, verbose = 2)

Summary of workflow

1. Tokenize tweets convert words to integers.
2. Filter short tweets split into training and test sets.
3. Pad sequences so all inputs have uniform length.
4. Build a neural network using word embeddings to represent words.
5. Train, visualize progress, and evaluate performance on unseen data.

In this project:

* Each tweet is converted into a sequence of word indices: “Trump wins election” [1, 2, 3]
* The **embedding layer** in Keras looks up each word’s vector from the embedding matrix.
* These vectors are **updated during training** to improve the model’s ability to distinguish between tweets made before or during Trump’s presidency.
* By the end, the embeddings capture meaningful information about the **context and relationships** of words in Trump’s tweets.

## Word Embedding Note

### The problem with text data:

Neural networks require **numeric input**, but words are **discrete symbols** like “Trump” or “tweet”.

A simple way to convert words to numbers is **one-hot encoding**, where each word gets a unique binary vector. \* E.g. with 5 words: “Trump” [1, 0, 0, 0, 0], “tweet” [0, 1, 0, 0, 0]. \* Drawbacks of one-hot encoding: - Vectors are very sparse (mostly zeros). - No sense of similarity between words — “president” and “leader” are just as different as “president” and “banana”. - Leads to high memory usage and inefficient learning.

### Solution: Word Embeddings

* Word embeddings map each word to a **dense, low-dimensional vector** of continuous numbers, like [0.2, -0.5, 0.8, …].
* These vectors are learned automatically by the neural network during training.
* Words with **similar meanings** end up having **similar vectors**, capturing semantic relationships.

How embeddings work internally:

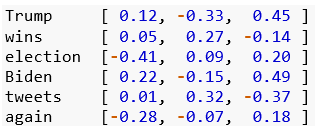
1. Imagine a matrix with:
   * Rows: one for each word in the vocabulary.
   * Columns: represent the embedding dimensions (features).
2. When a tweet is processed, each word is used to lookup its corresponding row in the matrix.
3. These vectors are then passed to the next layer of the network.
4. During backpropagation, the network updates the embedding values, learning which word features are useful for the task.

## Example

Suppose we have the following three tweets:

* Tweet 1: “Trump wins election”
* Tweet 2: “Biden wins election”
* Tweet 3: “Trump tweets again”

### 1. Vocabulary creation:

Unique words = [“Trump”, “wins”, “election”, “Biden”, “tweets”, “again”]

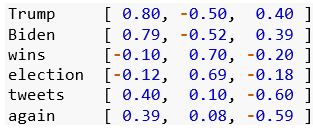
### 2. Initial random embeddings:

Let’s say we use an embedding dimension of 3. Initially, the embedding matrix might look like this:

### 3. Training process:

The network tries to predict whether a tweet was made before or after Trump’s presidency.

During training, backpropagation adjusts the values in this matrix so that meaningful relationships are captured:

* “Trump” and “Biden” might end up close together because both are political figures.
* “wins” and “election” may also cluster together.

### 4. Learned semantic meaning:

After training, the vectors could look like:

* “Trump” and “Biden” now have very similar vectors the model has learned they are related.
* “wins” and “election” are also close together, showing a learned association.

### Why embeddings are powerful

* Captures semantic relationships:
  + E.g. vector(“king”) - vector(“man”) + vector(“woman”) vector(“queen”).
  + This shows embeddings can model analogies and relationships between words.
* Efficient representation: a vocab of 10,000 words represented in a 100D embedding space uses far less memory than a 10,000-length one-hot vector.
* Generalizable knowledge: pre-trained embeddings (like Word2Vec or GloVe) can be used in new tasks, transferring learned word relationships to different models.

# 1D CNN trained from scratch

* This notebook continues using word embeddings as input features, similar to the previous notebook.
* The main change is in the neural network architecture, which is now a 1-Dimensional Convolutional Neural Network (1D CNN).
* A 1D CNN can capture relationships and patterns between neighboring words, such as common phrases or word sequences.
* The earlier steps, like loading data, tokenizing text, and creating embeddings, remain mostly the same as before.
* The focus of this notebook is on building and training the CNN model for tweet classification.

Tokenize the tweets:

max\_features <- 1000 # choose max\_features most popular words  
tokenizer = text\_tokenizer(num\_words = max\_features)  
fit\_text\_tokenizer(tokenizer, tweets\_sample$text)  
sequences = tokenizer$texts\_to\_sequences(tweets\_sample$text)

Remove tweets with less than 5 words:

y <- as.integer(tweets\_sample$is\_potus[seq\_ok])  
sequences <- sequences[which(seq\_ok)]  
tweets\_sample\_valid <- tweets\_sample[seq\_ok, ]

Split up the training and test set:

training\_rows <- which(tweets\_sample\_valid$id\_str %in% training\_ids$id\_str)  
  
train <- list()  
test <- list()  
train$x <- sequences[training\_rows]  
test$x <- sequences[-training\_rows]  
  
train$y <- y[training\_rows]  
test$y <- y[-training\_rows]

Pad the sequences:

maxlen <- 32   
x\_train <- train$x %>% pad\_sequences(maxlen = maxlen)  
x\_test <- test$x %>% pad\_sequences(maxlen = maxlen)

We can now define the model, again using 10 embedding dimensions.

embedding\_dims <- 10  
model <- keras\_model\_sequential() %>%   
 layer\_embedding(max\_features, output\_dim = embedding\_dims, input\_length = maxlen) %>%  
 layer\_dropout(0.2) %>%  
 layer\_conv\_1d(filters = 64, kernel\_size = 8, activation = "relu") %>%  
 layer\_max\_pooling\_1d(pool\_size = 2) %>%  
 layer\_flatten() %>%  
 layer\_dense(32, activation = "relu") %>%  
 layer\_dense(1, activation = "sigmoid")  
  
summary(model)

* MLP model:
  + layer\_flatten() flattens the embedding output directly into a 1D vector.
  + No understanding of local word patterns.
  + No pooling layer is used every feature from the embeddings is passed directly into the dense layers.
* CNN model: Adds a 1D convolution layer to detect local word patterns: layer\_conv\_1d()
  + Uses max pooling to reduce sequence length and focus on key features: layer\_max\_pooling\_1d()
  + Captures context and order of words (e.g. phrases like “fake news”).

The CNN is more powerful for text classification because it automatically extracts higher-level features from sequences of words, rather than relying solely on individual word embeddings.

Compile the model:

model %>% compile(  
 loss = "binary\_crossentropy",  
 optimizer = "adam",  
 metrics = "accuracy")

Train the model:

history <- model %>%  
 fit(x\_train, train$y,  
 batch\_size = 32, epochs = 10, verbose = 0)  
plot(history)

And evaluate the model on a test set:

results <- model %>% evaluate(x\_test, test$y, batch\_size=128, verbose = 2)

# Pre-trained embeddings

Change in embeddings approach: instead of learning word embeddings from scratch (random initialization), the model uses pre-trained embeddings.

* Pre-trained embeddings are vectors already trained on large text corpora, capturing rich semantic relationships between words.
* Transfer learning:
  + The embeddings are derived from a previously trained model, leveraging knowledge from large-scale datasets.
  + This approach allows the model to start with meaningful word representations, rather than learning from limited training data.
  + Can improve accuracy and convergence speed since the network doesn’t need to learn embeddings from scratch.
* Implementation with TensorFlow Hub:
  + TensorFlow Hub (TFHub) allows loading pre-trained models and embeddings easily.
  + Enables integration of embeddings like Word2Vec, GloVe, or BERT into a neural network.
* Benefit for text classification:
  + Pre-trained embeddings can help the model generalize better to unseen tweets.
  + They are particularly useful when the dataset is small or domain-limited, as in this case.

Use pre-trained embeddings leverage external knowledge feed into neural network improve performance without extensive training.

tweets\_sample\_valid <- tweets\_sample[seq\_ok, ]  
  
training\_rows <- which(tweets\_sample\_valid$id\_str %in% training\_ids$id\_str)  
  
train <- list()  
test <- list()  
train$x <- tweets\_sample\_valid$text[training\_rows]  
test$x <- tweets\_sample\_valid$text[-training\_rows]  
  
train$y <- as.integer(tweets\_sample\_valid$is\_potus[training\_rows])  
test$y <- as.integer(tweets\_sample\_valid$is\_potus[-training\_rows])

Now to build the model. This works similarly to the previous notebook, in that we are going to convert tweets into embeddings vectors. But now we use a pre-trained text embedding as the first layer, which has three advantages:

* You don’t have to worry about text preprocessing,
* Benefit from transfer learning,
* the embedding has a fixed size, so it’s simpler to process.

For this example you use a **pre-trained text embedding model** from [TensorFlow Hub](https://tfhub.dev) called [google/nnlm-en-dim50/2](https://tfhub.dev/google/nnlm-en-dim50/2).

There are many other pre-trained text embeddings from TFHub that can be used:

* [google/nnlm-en-dim128/2](https://tfhub.dev/google/nnlm-en-dim128/2) - trained with the same NNLM architecture on the same data as [google/nnlm-en-dim50/2](https://tfhub.dev/google/nnlm-en-dim50/2), but with a larger embedding dimension. Larger dimensional embeddings can improve on your task but it may take longer to train your model.
* [google/nnlm-en-dim128-with-normalization/2](https://tfhub.dev/google/nnlm-en-dim128-with-normalization/2) - the same as [google/nnlm-en-dim128/2](https://tfhub.dev/google/nnlm-en-dim128/2), but with additional text normalization such as removing punctuation. This can help if the text in your task contains additional characters or punctuation.
* [google/universal-sentence-encoder/4](https://tfhub.dev/google/universal-sentence-encoder/4) - a much larger model yielding 512 dimensional embeddings trained with a deep averaging network (DAN) encoder.

Find more about [text embedding models](https://tfhub.dev/s?module-type%20=%20text-embedding) on TFHub.

Let’s first create a Keras layer that uses a TensorFlow Hub model to embed the sentences, and try it out on a couple of input examples. Note that no matter the length of the input text, the output shape of the embeddings is: (num\_examples, embedding\_dimension)

embedding <- "https://tfhub.dev/google/nnlm-en-dim50/2"  
hub\_layer <- tfhub::layer\_hub(handle = embedding, trainable = TRUE)  
hub\_layer(train$x[1:2])

Now build the full model:

model <- keras\_model\_sequential() %>%  
 hub\_layer() %>%  
 layer\_dense(16, activation = "relu") %>%  
 layer\_dense(1, activation = "sigmoid")

The layers are stacked sequentially to build the classifier:

1. The first layer is a TensorFlow Hub layer. This layer uses a pre-trained Saved Model to map a tweet into its embedding vector. The pre-trained text embedding model that you are using ([google/nnlm-en-dim50/2](https://tfhub.dev/google/nnlm-en-dim50/2)) splits the sentence into tokens, embeds each token and then combines the embedding. The resulting dimensions are: (num\_examples, embedding\_dimension). For this NNLM model, the embedding\_dimension is 50.
2. This fixed-length output vector is piped through a fully-connected (Dense) layer with 16 hidden units.
3. The last layer is densely connected with a single output node.

We can now compile the model:

# Compile the model  
model %>% compile(  
 loss = "binary\_crossentropy",  
 optimizer = "adam",  
 metrics = "accuracy")  
  
# Train the model  
history <- model %>% fit(  
 train$x, train$y,  
 epochs = 10, batch\_size = 64, verbose = 0)  
plot(history)  
  
# Evaluate it's performance on a test set  
results <- model %>% evaluate(test$x, test$y, batch\_size = 128, verbose = 2)

# Creating a TensorFlow dataset

* Uses the same MLP model with pre-trained embeddings (google/nnlm-en-dim50/2).
* However, now we’re going to read in data in a different format, assuming that each document (each tweet, in our case) is in it’s own text file, and these files have been ordered in folders according to which class they are in (before or during tweets) and whether the are to be used for training or testing.
* TensorFlow can automatically infer labels from folder names.
* This structure is flexible, scalable, and easier to maintain, especially for large text datasets.

library(tfhub)

Dataset structure and preparation:

* Each tweet is stored in its own text file, allowing flexible organization.
* Files are organized into folders by:
  + Class: before vs during presidency.
  + Usage: train vs test.
* This structure allows TensorFlow to automatically infer labels from folder names.

Directory structure example:

main\_directory/  
 class\_a/  
 a\_text\_1.txt  
 a\_text\_2.txt  
 class\_b/  
 b\_text\_1.txt  
 b\_text\_2.txt

### Downloading and reading data

The following code downloads the dataset to your machine and unzips it:

download.file(  
 url = "https://github.com/iandurbach/datasci-fi/raw/master/data/trump\_tweets.zip?raw=TRUE",   
 destfile = "trumptweets.zip")  
  
unzip("trumptweets.zip", exdir = "data")

Note that each tweet is in its own file. Individual tweets can be read with readr::read\_file()

readr::read\_file("trump\_tweets/train/before/tr\_0\_1.txt")

Organizing tweets as separate files per class facilitates dataset creation and makes it easier to scale to new data.

### Creating a TensorFlow dataset

Now we can load the data off disk and prepare it for training. To do this, we use the [text\_dataset\_from\_directory](https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text_dataset_from_directory) function. This expects a directory structure where each class has its own folder. Each folder contains text files, one per document (tweet). TensorFlow automatically assigns labels based on folder names.

* Use text\_dataset\_from\_directory() to read data directly from directories.
* Parameters:
  + batch\_size: Number of examples per batch (e.g. 128).
  + validation\_split: Fraction of training data reserved for validation (e.g. 20%).
  + subset: Whether the dataset is “training” or “validation”.
* This function returns a TensorFlow Dataset object, where each element contains:
  1. A batch of tweets (text - the tweet is not preprocessed in any way).
  2. Corresponding labels (0 = before presidency, 1 = during presidency).

batch\_size <- 128  
seed <- 42  
  
train\_data <- text\_dataset\_from\_directory(  
 'data/trump\_tweets/train',  
 batch\_size = batch\_size,  
 validation\_split = 0.2,  
 subset = 'training',  
 seed = seed)  
  
validation\_data <- text\_dataset\_from\_directory(  
 'data/trump\_tweets/train',  
 batch\_size = batch\_size,  
 validation\_split = 0.2,  
 subset = 'validation',  
 seed = seed)  
  
test\_data <- text\_dataset\_from\_directory(  
 'data/trump\_tweets/test',  
 batch\_size = batch\_size)

Advantages of TensorFlow datasets

* Tweets are not preprocessed initially; preprocessing happens automatically when passing through the embedding layer.
* The dataset can be iterated batch-wise, making it memory-efficient and compatible with large corpora.
* Can be passed directly to Keras models, eliminating the need to manually split features (x) and labels (y).

Print the first 10 examples:

batch <- train\_data %>%  
 reticulate::as\_iterator() %>%  
 reticulate::iter\_next()  
  
batch[[1]][1]

Let’s also print the first 10 labels.

batch[[2]][1:10]

We can now proceed more or less as for the previous notebook. We first create a Keras layer that uses a TensorFlow Hub model to embed the sentences:

embedding <- "https://tfhub.dev/google/nnlm-en-dim50/2"  
hub\_layer <- tfhub::layer\_hub(handle = embedding, trainable = TRUE)

* Pre-trained embeddings from TensorFlow Hub are used (google/nnlm-en-dim50/2).
* layer\_hub(handle = embedding, trainable = TRUE):
  + Loads the embedding model.
  + Embeddings are fine-tuned during training if trainable = TRUE.
* Each tweet (text string) is converted into a dense vector representation capturing semantic meaning.

And then build the full model:

model <- keras\_model\_sequential() %>%  
 hub\_layer() %>% # converts raw text into embeddings  
 layer\_dense(16, activation = 'relu') %>%  
 layer\_dense(1, activation = 'sigmoid')

We can then configure and compile the model:

model %>% compile(  
 loss = "binary\_crossentropy",  
 optimizer = "adam",  
 metrics = "accuracy")

And then train and evaluate it. Notice the slightly different format used to specify the input data – because we have created a TensorFlow dataset train\_data and test\_data, we can pass them in directly, rather than having to specify the features (previously, train$x) and outcomes (previously, train$y) explicitly.

history <- model %>% fit(train\_data, epochs = 10, verbose = 0)  
plot(history)

results <- model %>% evaluate(test\_data, verbose = 2)

### Key concepts

* Pre-trained embeddings: Leverage external knowledge, improving performance on small datasets.
* TensorFlow Dataset API: Efficient, batch-wise data feeding for training neural networks.
* Directory-based labeling: Simplifies dataset creation and ensures consistent label assignment.
* Transfer learning: Fine-tuning pre-trained embeddings allows the network to adapt to the specific task (Trump tweet classification).