Doc2Vec

Thomas Asikis

11/27/2019

Introduction

In this example we will try to embed sequences of words instead of just words. To do so we will us as representation the word2vec word embeddings created in the previous example.

Main Libraries

Here are the main libraries that the scripts below rely on:

```
library(keras)

library(stringi)
library(stopwords)
library(tokenizers)
library(pbapply)
library(wordVectors)
library(magrittr)
library(data.table)
library(purrr)
```

```
##
## Attaching package: 'purrr'
```

```
## The following object is masked from 'package:data.table':
##
##     transpose
```

```
## The following object is masked from 'package:magrittr':
##
##     set_names
```

```
library(umap)
library(ggplot2)
library(Rtsne)
library(tidyr)
```

```
##
## Attaching package: 'tidyr'
```

```
## The following object is masked from 'package:magrittr':
##
##     extract
```

Data Loading

First we load the parliamentary corpus by downloading it from: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/6MZN76 and storing it locally. The corpus focuses on parliamentry debates

of the Irish parliament. We repeat the same preprocessing steps as in the word embedding example, only here we pick a smaller of documents to process, as the models tend to be very expensive. In general, processing a lot of documents takes a lot of time in R without optimizations. For example 60k documents take an hour to be matched to sequence embeddings.

```
# source.folder <- setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
corpus <- data.table::fread("Dail_debates_1919-2013.tab", header = T)
corpus$date <- as.Date(corpus$date)
last_goverment_begin_data <- as.Date('2011-03-09')
corpus <- corpus[corpus$date >= last_goverment_begin_data, ]

# subset to process, indices picked randomly
corpus <- corpus[100:1100, ]
corpus$norm_speech <-  stri_trans_general(str = corpus$speech,
                                          id = "Latin-ASCII")
# sentence tokenization
corpus$sentences <-  pblapply(corpus$norm_speech, tokenize_sentences, simplify=TRUE)
corpus$n.sentences <- sapply(corpus$sentences, length)

# keep all speeches with more than one sentence,
# as we plan to train sentence to sentence models
corpus <- corpus[corpus$n.sentences > 1, ]
```

Preprocessing

Now we prepare the sequence embeddings. The goal is to match each word of a sentence to each representation from the word2vec model. So each sentence would now be a matrix of dimensions $words\_in\_sentence \times embedding\_dimensions$.

Match word to word embedding

Since we need to clarify to our model the beginning and end of each sentence, we create 2 special tokens that are matched to unique embeddings. If we don't have an embedding for a word we ommit it. This happens because the word was not included in the corpus we trained word2vec. Other strategies can also be considered, e.g. assign the word to a random vector, retrain word embeddings etc. So first we need to load the word reprentations and prepare a function that matches them to words.

```
## Filename ends with .bin, so reading in binary format
```

```
## Reading a word2vec binary file of 34934 rows and 50 columns
```

Match sequence to word embeddings matrix

Now we prepare a function that takes a word sequence, breaks it into words and then matches each word to each embedding.

Document Embeddings

Now we have our data loaded and each sentence is mapped to a collection of vectors. Our next goal is to get a single embedding (a vector of 50 numbers) for each sentence.

Average Word Embeddings

Sentence emebeddings can be calculated as aggregations of the word embeddings in each sentece. For a first illustration we showcase the easiest way to do this, e.g. by averaging all the word embeddings in a sentence:

Here is how the sentence embedding of the first sentence of the first speech of our corpus looks like:

```
corpus$sentence_embeddings[[1]][[1]]
```

```
##  [1]  0.095605808  0.279836495  0.099741940  0.237374297  0.323085921
##  [6]  0.077296346  0.254031879  0.002553458  0.142707169  0.268106592
## [11]  0.236565413  0.182779763  0.113539524  0.236924317  0.093921208
## [16]  0.360592365  0.474872375  0.028019176  0.088001532 -0.011190522
## [21]  0.390843183  0.299049675  0.365114619  0.151107465  0.348743424
## [26]  0.004588029  0.171405226  0.188500637  0.267830661  0.402600324
## [31]  0.275389290  0.304002381  0.152752622  0.284913830  0.059937598
## [36]  0.325035284  0.206012902  0.157317960  0.402527466  0.062888602
## [41]  0.548809251  0.116637352  0.117901985  0.163983005  0.379096663
## [46]  0.118640418  0.017552439  0.185591939  0.420274252  0.049532239
```

Since we use average we can use projections on the same dimensions of senteces, words and speeches. We could even expand the above logic to politicians or parties with some group_by operations and then we could use a distance based approach to generate clusters via clustering or nearest neighborhs for network plots.

Now we use the same average logic to extend to speech embeddings, were we average over all the sentence embeddings in a speech.

```
# function that averages over a list of sentence embeddings
average_over_speeches <-function(sentences_embs){
  sentences_embs <- as.matrix(do.call(cbind, sentences_embs))
  speeches_average <- rowMeans(sentences_embs)
  return(speeches_average)
}

corpus$speech_embeddings <- pbsapply(corpus$sentence_embeddings, average_over_speeches, simplify=FALSE)
```

A speech embedding looks like:

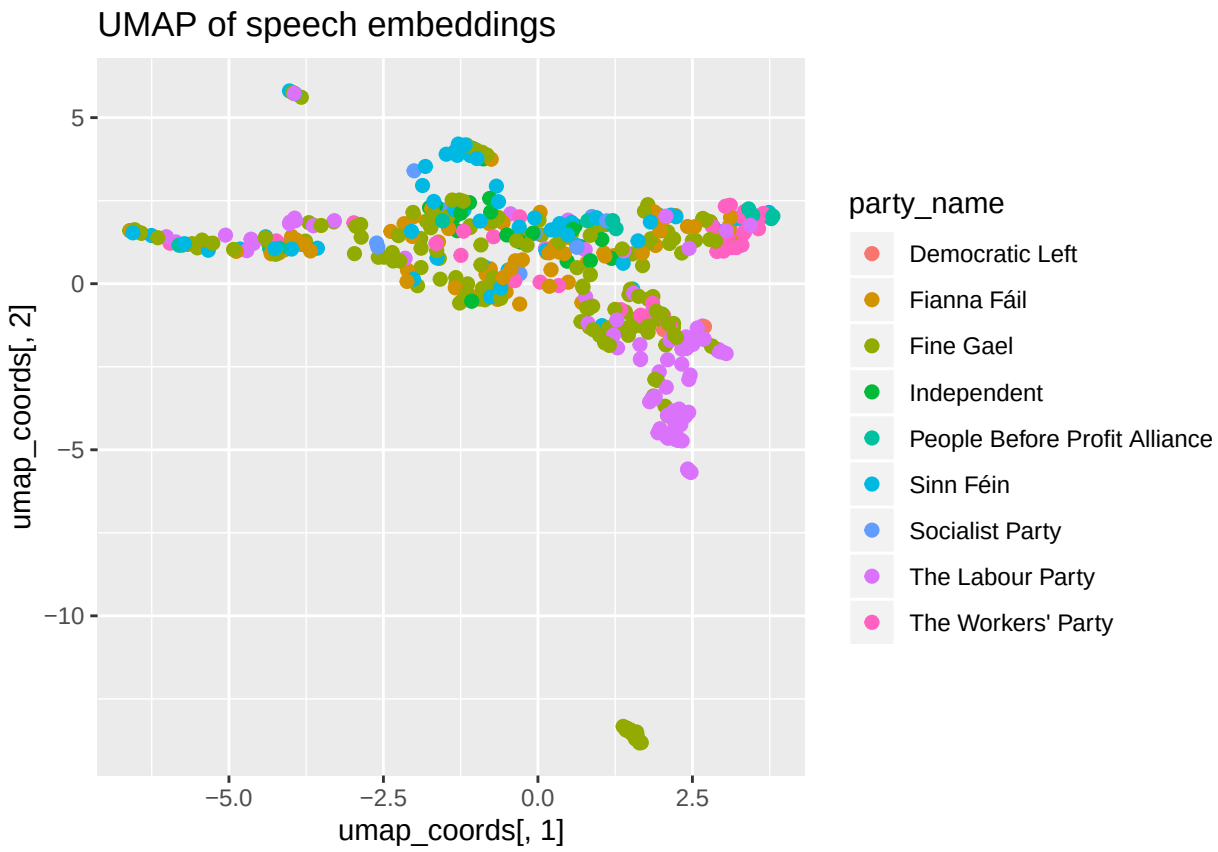
```
corpus$speech_embeddings[[1]]
```

```
##  [1] -0.004361339  0.095746242  0.100113320  0.072919646  0.204525736
##  [6] -0.044233137  0.236594215 -0.162703979 -0.027113403  0.261865672
## [11]  0.218814600 -0.007146483 -0.011524459  0.108228626  0.055968576
## [16]  0.301686122  0.350727180 -0.070690766 -0.003386216 -0.020434291
## [21]  0.207580314  0.306156047  0.253566575  0.134415733  0.248558092
## [26] -0.063832669 -0.039650680  0.180309104  0.198793722  0.334334460
## [31]  0.181872278  0.232151924  0.109343512  0.216617858  0.088348047
## [36]  0.268132025  0.202590449  0.114905952  0.450052642 -0.136539776
## [41]  0.359541300  0.149722713 -0.003766483  0.155250864  0.214060321
## [46]  0.109545437  0.045693294  0.172679942  0.314497452  0.045655036
```

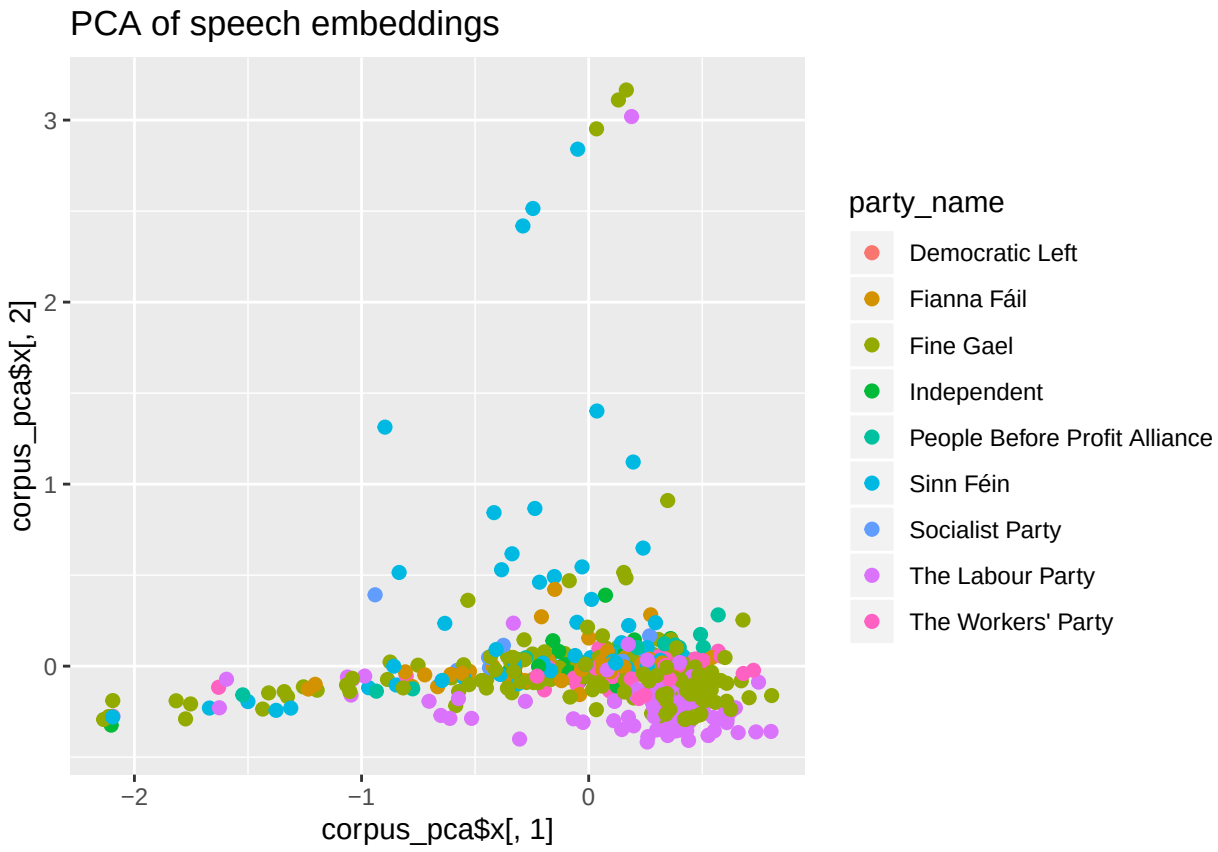
And now let's visualize the speeches as points after a dimensionality reduction. Each speech is colored according to the party the speaker belongs to. If word embeddings are working as expected and averaging over them has the desired effect, then points that are close in the plots below, are also considered close contextually.

```
# Now some visualizations
# A rich collection of visualization is found at:
# https://www.analyticsvidhya.com/blog/2017/01/t-sne-implementation-r-python/
```

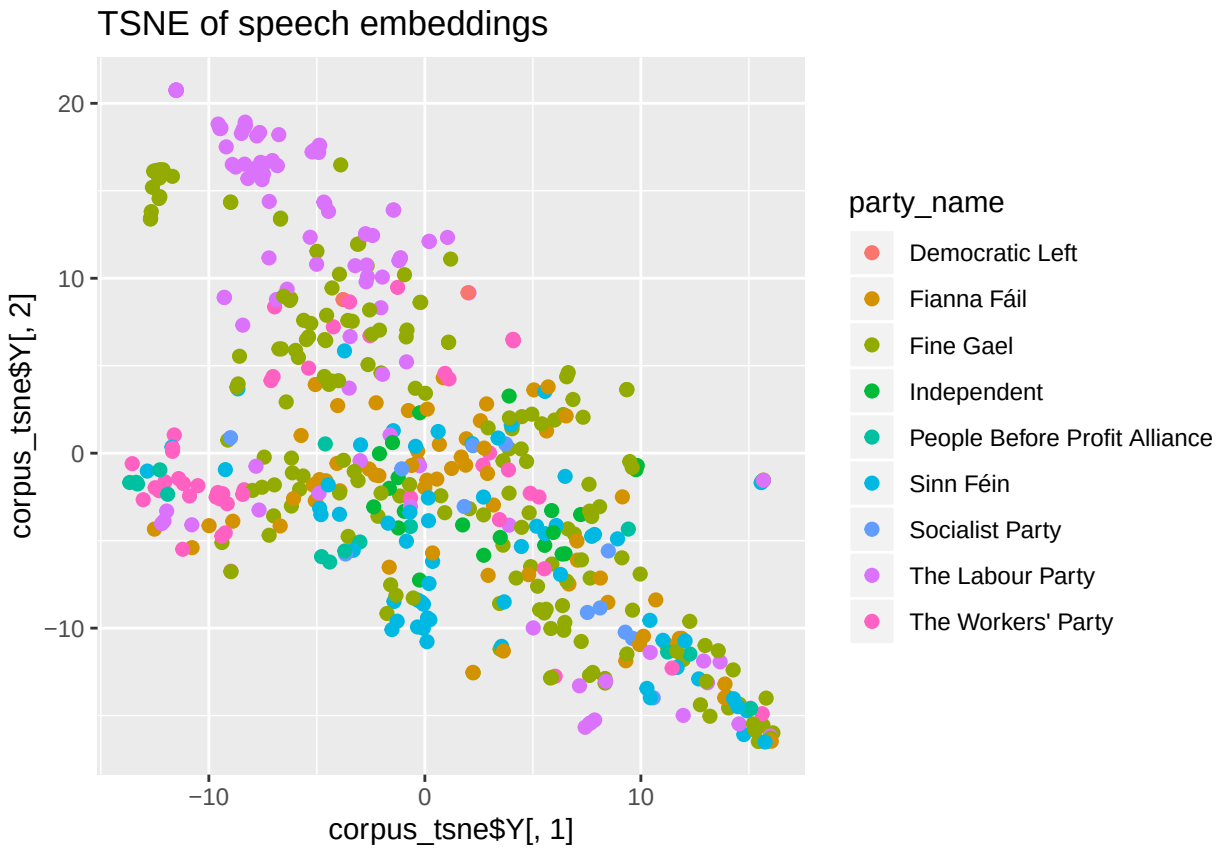
```
custom.config = umap.defaults
custom.config$random_state = 123

umap_embs <- umap(do.call(rbind, corpus$speech_embeddings), custom.config)
umap_coords <- umap_embs$layout
corpus$speeches_x <- umap_coords[, 1]
corpus$speeches_y<- umap_coords[, 2]
ggplot(corpus, aes(x= umap_coords[, 1] , y=umap_coords[, 2], color=party_name)) +
  geom_point(size=2) +
  ggtitle("UMAP of speech embeddings")
```



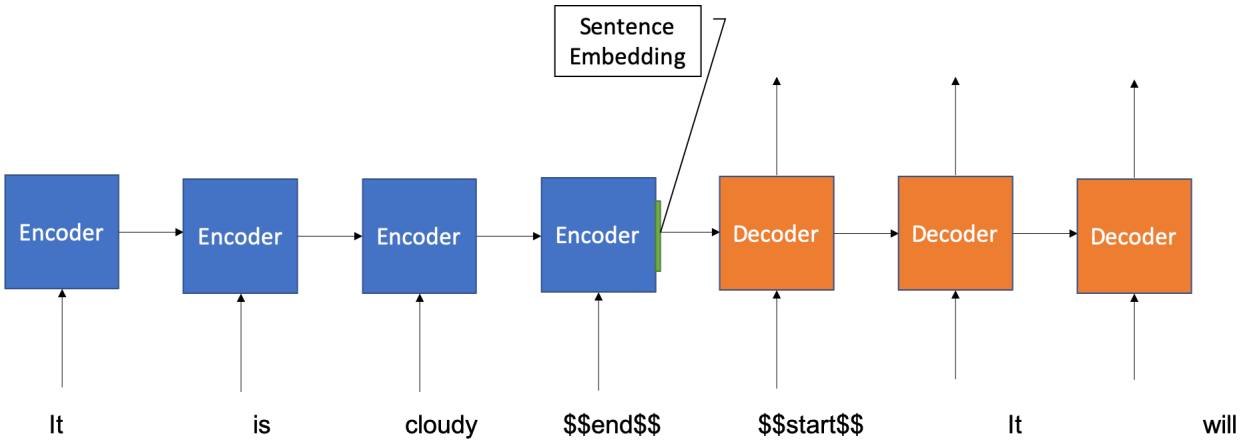

```
corpus_pca <- prcomp(do.call(rbind, corpus$speech_embeddings),center = TRUE)
ggplot(corpus, aes(x= corpus_pca$x[, 1] , y=corpus_pca$x[, 2], color=party_name)) +
  geom_point(size=2) +
  ggtitle("PCA of speech embeddings")
```




```
corpus_tsne <- Rtsne(do.call(rbind, corpus$speech_embeddings), dims = 2,
                     perplexity=30, verbose=FALSE, max_iter = 500, check_duplicates = FALSE)
ggplot(corpus, aes(x=corpus_tsne$Y[,1], y=corpus_tsne$Y[,2], color=party_name)) +
  geom_point(size=2) +
  ggtitle("TSNE of speech embeddings")
```



Sequential Embeddings Let's use a more sophisticated model to calculate the sentece embeddings. We now use a sequence to sequence model, that takes a sentence word embedding as input and tries to predice the word embedding of the next matrix. This way context also included in a sequential manner and not as a bag of words (as in the average solution). The main logic of sequence embeddings relies on the notion that input sentences that yield the same embedding are expected to have similar sentences succeeding them. Therefore, one may assume that the context of preceeding sentences, which have low distance between theis embeddings, is similar in terms that it leads to same contextual statements after them. In general the model looks like:



Preparing the sequential data For training the goal is to create a matrix of senteces word embeddings for the sentence we have in the corpus. This matrix will be used to create pair of consecutive sentences, so that the model can predict one sentence based on the previous one. To fit all sentences in the same matrix, we have to pad them in the same size. This means that we need to find the sentence with most word embeddings in its matrix. Then attach zero values word emebddings to each sentence matrix with less elements, untill it has the same number of word embeddings as the max element matrix. If in the test set we observer an even

bigger sentence, then the model will still work. So we don't have to prune the extra words. The main goal of padding during training is to allow batch training which improves speed and often generalization.

```
# pad sentence matrix
pad_sentence_word_embeddings<-function(sentence_word_embedding){
  pad_differences <- max_sequence_length - nrow(sentence_word_embedding)
  if(pad_differences > 0){
    pad_matrix <- matrix(0, nrow = pad_differences, ncol =word_embedding_size)
    # here I use a zero filled matrix for speed, which is the same as adding the end sequence
    # embedding many times. Some people differentiate padding (always zeros), with end embeddings (coul
    # other people train word embedding models with sequence begin/end tokens.
    # we should be careful when doing so, because such tokens behave like stop-words (they are very fre
    sentence_word_embedding <- do.call(rbind, list(sentence_word_embedding, pad_matrix))
  }
  return(sentence_word_embedding)
}
example_pad <- pad_sentence_word_embeddings(corpus$sentence_word_embeddings[[1]][[1]])

# match consecutive sentence pairs and pad each sentence once
get_sentence_pairs <- function(sentence_word_embs){
  pairs <- list()
  prev_embedding <- pad_sentence_word_embeddings(sentence_word_embs[[1]])
  for(k in 2:(length(sentence_word_embs))){
    next_embedding <-  pad_sentence_word_embeddings(sentence_word_embs[[k]])
    first_sentences[[sample_counter]] <<- prev_embedding
    second_sentences[[sample_counter]] <<- next_embedding
    prev_embedding <- next_embedding
    sample_counter <<- sample_counter + 1
  }
}

# prepare input and output data
sample_counter <- 1
first_sentences <- list()
second_sentences <- list()

lapply(corpus$sentence_word_embeddings, get_sentence_pairs)

list_to_array<-function(element_list){
  # since keras works with arrays, here is an implementation that
  # converts a list of matrices to a 3-D array.
  # For more complex models, you can use more dimensions.
  # the input of sequential models is 3-D:
  # number of examples (batch size) X sequence steps X num_features
  element_list <-  array(unlist(element_list), dim = c(length(element_list),
                                                       nrow(element_list[[1]]),
                                                       ncol(element_list[[1]])))
  return(element_list)
}

first_sentences <- list_to_array(first_sentences)
encoder_input_data <- first_sentences
second_sentences <- list_to_array(second_sentences)
# predict the next sentence given the prevous sentence and a word at a time
```

```
# so the input to the decoder includes every word except the last one (end token/pad)
decoder_input_data <- second_sentences[, 1:dim(second_sentences)[2]-1, ]
# and so the target data are always one word ahead of the decoder input data.
target_data = second_sentences[, 2:dim(second_sentences)[2], ]
```

Creating the model

To create the model we do the following:

```
# Prepare sequence model
# since this model outputs 2 states per sentence,
# we use half the word embedding size here
# and concatenate the state vectors in one:
sequence_embedding_size = 25

# Define an input sequence and process it.
# here null is the number of words per sentece,
# and since it is not fixed, null means dynamic
encoder_inputs  <- layer_input(shape=list(NULL, word_embedding_size))
encoder         <- layer_lstm(units=sequence_embedding_size, return_state=TRUE)
encoder_results <- encoder_inputs %>% encoder
# We discard `encoder_outputs` and only keep the states.
# lstm produces 2 states, other sequential models may produce more or less states
# we may decide to keep all or a subset of the states and we usually concatenate them
# still, this depends on the model and requires a bit of reading to have an educated guess.
encoder_states  <- encoder_results[2:3]

## Set up the decoder, using `encoder_states` as initial state.
decoder_inputs  <- layer_input(shape=list(NULL, word_embedding_size))
## We set up our decoder to return full output sequences,
## and to return internal states as well. We don't use the
## return states in the training model, but we will use them in inference.
decoder_lstm  <- layer_lstm(units=sequence_embedding_size, return_sequences=TRUE,
                            return_state=TRUE, stateful=FALSE)
decoder_results <- decoder_lstm(decoder_inputs, initial_state=encoder_states)

## here we use a linear activation. if we used some kind of binary representation
## the activation would be softmax, so that the output resembles a probability distribution
## over words.
decoder_dense   <- layer_dense(units=word_embedding_size, activation='linear')
decoder_outputs <- decoder_dense(decoder_results[[1]])

# Define the model that will turn
# `encoder_input_data` & `decoder_input_data`
# into `decoder_target_data`
model <- keras_model( inputs = list(encoder_inputs, decoder_inputs),
                      outputs = decoder_outputs )

# Compile model
model %>% compile(optimizer=optimizer_rmsprop(lr = 0.01, rho = 0.9, epsilon = 0.0001, decay = 0.01) , l
## for classification/binary emebddings: model %>% compile(optimizer='rmsprop', loss='categorical_cross

## Run model
```

```
model %>% fit( list(encoder_input_data, decoder_input_data), target_data,
               batch_size=64, # number of sentences to train in per time
               epochs=3, # total passes over all dataset

               validation_split=0.2 # a validation set to report error on
               )

## Save model
# save_model_hdf5(model,'s2s.h5')
# save_model_weights_hdf5(model,'s2s-wt.h5')
```

Now we just want the first part of the model to encode first sentences and use the embedding. As the model is trained we can create a "submodel" from the first part:

```
encoder_model <-  keras_model(inputs = encoder_inputs, outputs = encoder_states)
# to save the model
# save_model_hdf5(model,'encode_model.h5')
# save_model_weights_hdf5(model,'encode_model-wt.h5')
```

Let's now see how an lstm embedded sentece looks like:

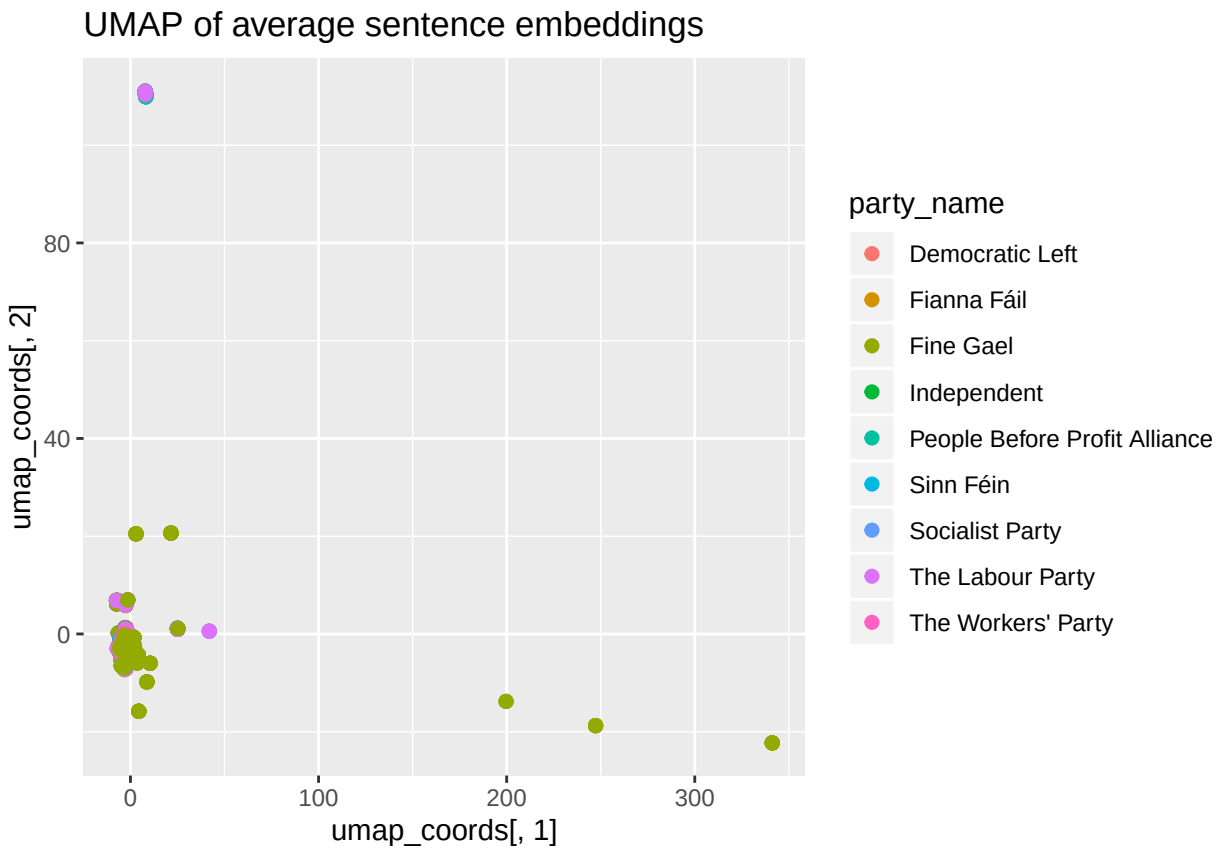
```
## get the emebedding for a new sentence:
new_sentence <- "$$begin$$ I always wanted to spend all the goverment funds for education $$end$$"

#Now we have to apply all preprocessing steps to this sentece
embedded_new_sentence <- embed_sequence(new_sentence)

# function to get sequential embedding
lstm_embed_sentence<-function(sentence){
  embedded_sentence <- embed_sequence(sentence)
  padded_embedded_sentece <- pad_sentence_word_embeddings(embedded_new_sentence)
  emb <- encoder_model %>% predict(list_to_array(list(padded_embedded_sentece)))
  return(unlist(emb))
}

lstm_embed_sentence(new_sentence)
```

```
##   [1]  1.122670e-02 -1.220957e-02  7.134781e-03  9.678536e-03 -1.073443e-02
##   [6]  6.799669e-03 -1.648001e-03 -1.038811e-02 -1.623048e-02 -1.896780e-02
##  [11]  2.232128e-02 -1.020162e-07  1.234287e-03  1.460234e-02 -1.811777e-03
##  [16] -5.673749e-03 -5.969035e-03  7.151290e-03 -1.284411e-02 -1.412006e-02
##  [21]  2.009595e-02 -2.470209e-03 -3.138380e-03  5.317368e-03 -1.040657e-02
##  [26]  2.244325e-02 -2.438911e-02  1.429417e-02  1.935929e-02 -2.154410e-02
##  [31]  1.357645e-02 -3.295095e-03 -2.070145e-02 -3.250184e-02 -3.798292e-02
##  [36]  4.482616e-02 -2.041490e-07  2.473993e-03  2.921967e-02 -3.604758e-03
##  [41] -1.139786e-02 -1.194711e-02  1.433691e-02 -2.559587e-02 -2.822103e-02
##  [46]  4.017206e-02 -4.944744e-03 -6.261842e-03  1.069982e-02 -2.080033e-02
```

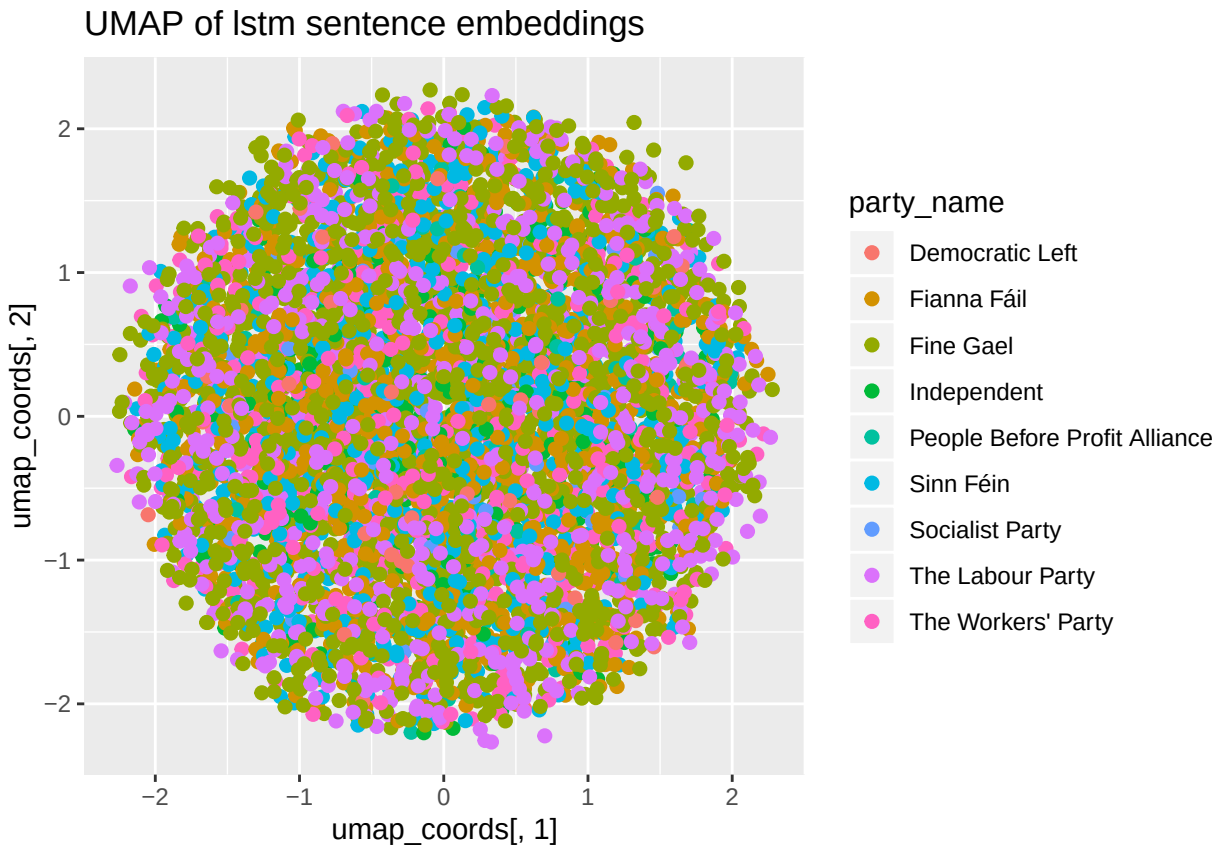
And now let's look at some UMAP visualizations of sentences per party given the sequential model and the average:

```
extended_corpus <- corpus %>%
  tidyr::unnest(c('sentences', 'sentence_embeddings'))
custom.config = umap.defaults
custom.config$random_state = 123
##model <- load_model_hdf5('s2s.h5')
```

```
##load_model_weights_hdf5(model,'s2s-wt.h5')
umap_embs <- umap(do.call(rbind, extended_corpus$sentence_embeddings), custom.config)
umap_coords <- umap_embs$layout
ggplot(extended_corpus, aes(x= umap_coords[, 1] , y=umap_coords[, 2], color=party_name)) +
  geom_point(size=2) +
  ggtitle("UMAP of average sentence embeddings")
```




```
# lstm embeddings

res <- pbsapply(extended_corpus$sentences, lstm_embed_sentence)
res<-unname(res)
res_matrix <- t(as.matrix(res))

umap_embs <- umap(res_matrix, custom.config)
umap_coords <- umap_embs$layout
ggplot(extended_corpus, aes(x= umap_coords[, 1] , y=umap_coords[, 2], color=party_name)) +
  geom_point(size=2) +
  ggtitle("UMAP of lstm sentence embeddings")
```


UMAP of lstm sentence embeddings
umap_coords[, 2]
umap_coords[, 1]
2
1
0
−1
−2
−2
−1
0
1
2
party_name
Democratic Left
Fianna Fáil
Fine Gael
Independent
People Before Profit Alliance
Sinn Féin
Socialist Party
The Labour Party
The Workers' Party