Workshop ‘Computational Text Analysis’

Session 3: From Word Embeddings to Transformer Models

Mirko Wegemann

26 March 2025

Goals of this workshop

1. Automation of data collection
2. **Analysis of textual data**
   - unsupervised approaches (e.g., topic models)
   - supervised approaches (e.g. text classification)
3. Analysis of images-as-data

What we'll cover today

- We deal with a more advanced representation of text by looking at a multidimensional embedding space
- We use embeddings to understand the context in which certain terms are used and how this context may change over time
- We also use embeddings to apply deep neural networks in supervised classification
- We (may) get an idea of how embeddings relate to transformer models and how we could apply these in Python

Why bags-of-words may not be sufficient...

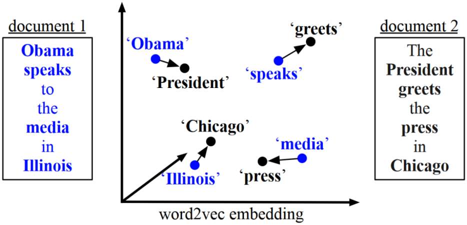
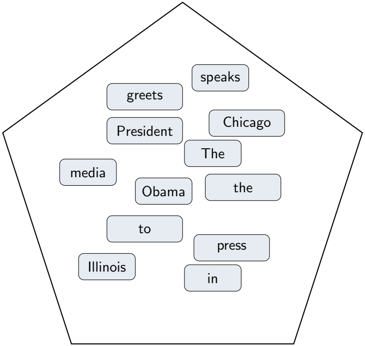


Figure 1. An illustration of the *word mover's distance*. All non-stop words (***bold***) of both documents are embedded into a *word2vec* space. The distance between the two documents is the minimum cumulative distance that all words in document 1 need to travel to exactly match document 2. (Best viewed in color.)

Kusner et al. (2015)

What bags-of-words would do... I

Remember a **bag-of-words**



What bags-of-words would do... II

```
> dfm_example
Document-feature matrix of: 2 documents, 12 features (37.50% sparse) and 0 docvars.
       features
docs    obama speaks to the media in illinois . president greets press chicago
  text1     1      1  1   1     1  1        1 1         0      0     0       0
  text2     0      0  0   2     0  1        0 1         1      1     1       1
>
```

This is how our text would look in a bags-of-words structure

What could be problematic here?

What bags-of-words would do... III

Major shortcoming of bags-of-words approaches is that they neither capture the (1) **relation of a word towards other words** nor the (2) **position of a word** within a sentence.

They are context-blind.

What bags-of-words would do... IV

We can distinguish into two types of embeddings:

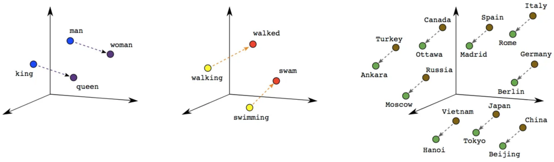
1. static embeddings, like GloVe (Pennington et al. 2014) or numberbatch
2. contextual embeddings, like Bert (Peters et al. 2018)

The idea behind word embeddings I

- idea: capture the similarity of words
- we can assign a word to a **multidimensional vector** that denotes its relationship towards other words
- "distances between such vectors are informative about the **semantic similarity** of the underlying concepts they connote for the corpus on which they were built" (P. Rodriguez and Spirling 2021)
- they can "predict the occurrence of a word by the surrounding word in a text sequence" (Rheault and Cochrane 2020, p. 112)

The idea behind word embeddings II

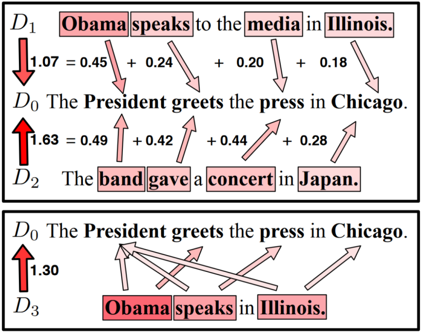
Embeddings put words in a k-dimensional space (simplified to three dimensions here).



Male-Female Verb Tense Country-Capital

1

The idea behind word embeddings III



Kusner et al. (2015)

[1] Source: https://towardsdatascience.com/a-guide-to-word-embeddings-8a23817ab60f

For what they may be helpful...

P. Rodriguez and Spirling (2021) distinguish between two functions:

1. instrumental function: they capture more meaning than bags-of-words → they could enhance our classification tasks
2. they have a value in themselves (analysis on the word level)
    - for example: if the distance between the term 'migrant' and 'hardworking' is closer for Green than for Conservative parties, we might infer underlying party framing strategies

Decisions to make when using word embeddings I

- **word2vec** vs. **GloVe** vs. **fasttext** vs. **numberbatch** (difference in training of embeddings)
- **pre-trained** vs. **locally trained** embeddings
- **window size**: how large is the context of surrounding words
- **dimensionality**: how many dimensions do we consider

Decisions to make when using word embeddings II

Pre-trained vs. local

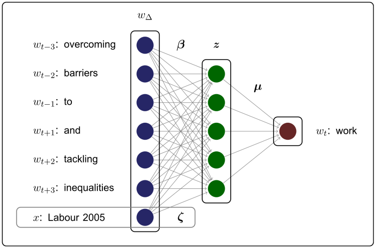
- representations of a word usually learned by deep learning techniques (like neural networks)
- we can either create corpus-specific embeddings which learn the relations only from a specific domain
- ...or use pre-trained embeddings (often trained on a large corpus like the Wikipedia)
- decision depends on corpus size and specificity

Decisions to make when using word embeddings III

Window size

- the larger the window size, the more distant words will influence the vector of an embedding

We predict a word w_t by the words surrounding it w_{t-1}, w_{t-2}, ..., w_{t-n} $\rightarrow$ n is the window size



Rheault and Cochrane (2020, p. 116)

Decisions to make when using word embeddings IV

Dimensionality

- the number of dimensions can be freely chosen
  - the term *dimensions* refers to the length of a term's vector representation
  - the more dimensions, the more we features of a word we capture but the more computationally intense it becomes (and the more models tend to overfit)

Decisions to make when using word embeddings V

In practice, hyperparameter choice only has marginal effects – pre-trained models perform well (P. Rodriguez and Spirling 2021)

Using word embeddings I

There are three different methods to obtain embeddings

1. train your own "local" embeddings with a neural network
2. take pre-trained embeddings (numberbatch, e.g., for GloVe or other multilingual embeddings)
3. fine-tune pre-trained embeddings

Using word embeddings II

Depending on the method you choose, different steps are involved.
For **pre-trained embeddings**:

- Data pre-processing
- Match between input and embedding data
- Data analysis

For **locally-trained embeddings**:

- Data pre-processing
- Use a neural network to learn the embeddings
- Data analysis

We'll focus on pre-trained embeddings (which we later fine-tune) here, but there's some code in the script on training your own embeddings.

In R: Pre-processing

We do not necessarily need to pre-process but:

- removing very frequent words (or stopwords) without much semantic meaning may boost our embeddings
- same applies to characters (like punctuation and often digits)
- more advanced: collocation analysis creating pairs of common phrases (e.g. European_Union) or lemmatization

Tokenization for embeddings

For embeddings, we'll use a different tokenizer that creates a slightly different data structure compared to common tools in `quanteda` which are used for bags-of-words approaches.

```
> tokens <- word_tokenizer(df$text_prep)
> it <- itoken(tokens)
> vocab <- create_vocabulary(it, stopwords =
    c(stopwords(language="en"),"also", "s", "t", "d"))
> tail(vocab)
Number of docs: 187689
179 stopwords: i, me, my, myself, we, our ...
ngram_min = 1; ngram_max = 1
Vocabulary:
      term  term_count doc_count
1:   national      25786     21412
2:     ensure      27085     23822
```

Match between input and embedding data

If we use pre-trained embeddings, it's possible that some words are not part of the embedding matrix and vice versa. We need to align both matrices.

1. Identify words not in the pre-trained embeddings:

```
not_in_emb <- vocab %>%
filter(!term %in% emb_wts$V1)
```

Match between input and embedding data II

2. Drop terms not in our corpus

```
# store V1 (terms) as row names
row_names <- emb_wts %>%
filter(V1 %in% vocab$term)  %>%
select(V1)

# filter the embeddings to terms of our corpus
embeddings <- emb_wts %>%
filter(V1 %in% vocab$term) %>%
select(-V1) %>%
as.matrix()

# set terms as rownames
rownames(embeddings) <- row_names$V1
```

Match between input and embedding data III

3. Add terms not in pre-trained corpus

```
# add those words which embedding vector does not have
    with a 0
embeddings_na <- matrix(data = 0, nrow =
    nrow(not_in_emb), ncol = 300)

# set terms as rownames
rownames(embeddings_na) <- not_in_emb$term

# row bind available and not available embeddings
embeddings <- rbind(embeddings, embeddings_na)
```

Let's do it in R

Descriptive analysis I

Nearest Neighbours (which words are close to each other?)
A traditional example is the equation

$$\mathit{Berlin} = \mathit{Paris} - \mathit{France} + \mathit{Germany} \tag{1}$$

Basically saying that the distance between the word Berlin and Germany should be the same as the one between Paris and France.

What's the capital of Germany?

We can translate this equation to R.

```
> which_capital = embeddings["paris", , drop = FALSE] -
+     embeddings["france", , drop = FALSE] +
+     embeddings["germany", , drop = FALSE]
> capital_cos_sim = sim2(x = embeddings, y = which_capital, method = "cosine", norm = "l2")
> head(sort(capital_cos_sim[,1], decreasing = TRUE), 5)
berlin   germany frankfurt   hamburg     paris
0.7635345 0.7232592 0.6718858 0.6530555 0.6509711
```

More substantively

Terms in proximity to "migrant"

```
> find_nns(embeddings['migrant',], pre_trained =
    embeddings, N = 20)
[1] "migrant"      "immigrant"    "refugee"
    "worker"       "undocumented" "farmworker"
    "indigenous"
[8] "unskilled"    "migration"    "expatriate"
    "immigration"  "migratory"    "resettlement" "labor"
[15] "employment"   "unemployed"   "population"
    "plight"       "welfare"      "labour"
```

Descriptive Analysis II

Similarities between terms by grouping variables (which words are used by which actors?)

```
context_wv_parfam <- dem_group(context_dem, groups =
    context_dem@docvars$parfam)
dim(context_wv_parfam)

context_nns <- nns(context_wv_parfam, pre_trained =
    embeddings, N = 10, candidates =
    context_wv_parfam@features, as_list = TRUE)
context_nns
```

Let's do it in R

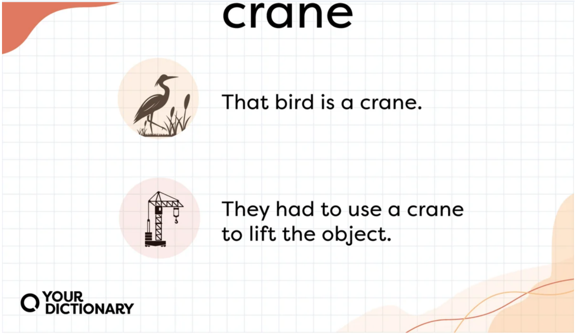
Embedding regression I

- **contextual use** of a term
- for this, we use covariates (as before in the descriptive analysis)
- we basically **average an embedding** in different contexts and compare how similar it is (before we attribute low weights to words which often occur in both contexts)

P. L. Rodriguez et al. (2023)

Embedding regression II

As we take the context into consideration, we come closer to capturing different word meanings:



Embedding regression in R

conText allows you to estimate similarities by group for every target term and provides you with standard errors

```
set.seed(451)
model2 <- conText(formula = trump ~ prepost,
data = toks,
pre_trained = embeddings,
transform = TRUE, transform_matrix = trans_mat,
bootstrap = TRUE,
num_bootstraps = 100,
permute = TRUE, num_permutations = 10,
window = 10,
verbose = T)
```

Let's do it in R

Evaluation of Embeddings I

- for downstream tasks (e.g., classification with embeddings), we can use typical metrics in machine learning (F1, accuracy, confusion matrix, etc.)
- for intrinsic tasks, more difficult: what makes an embedding particularly good?

Evaluation of Embeddings II

P. Rodriguez and Spirling (2021) suggest ‘turing validation’

- computing tasks good if they cannot be differentiated from human tasks
- in practice: let humans compare a list of word similarities made by humans with a list by embeddings

$\rightarrow$ if humans cannot distinguish between both lists, performance is good

- other validation includes correlation between models
- in the scaling task by Rheault and Cochrane (2020), cross-validation with other measures of party positions

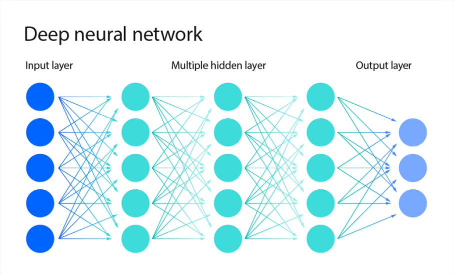
Time for a break?

Embeddings and deep learning

"Embeddings are numerical representations of real-world objects that machine learning (ML) and artificial intelligence (AI) systems use to understand complex knowledge domains like humans do."
Amazon AWS

- they reduce dimensionality and provide more information than bags-of-words
- in addition to instrumental interests in similarities, they can form the starting point for downstream tasks like classification

Excursion: Neural Networks I



In neural networks, we do not tell the network how to make sense of our data but let it learn
Introduction to deep learning

Excursion: Neural Networks II

- neural network has several **layers** that connect input to output data
- the intermediate layers are so-called '**hidden layers**'

→ there are always **one** input and **one** output layer; but there can be several hidden layers

Excursion: Neural Networks III

- **cost function**: deviation between predicted and actual values $\rightarrow$ error
- split in **training** and **test** data: to monitor predictive quality [optimally, even threefold split into training, test and evaluation data]
- **deep learning**: neural network with more than 3 layers (more than one hidden layer)

Excursion: Neural Networks III

Function of neural networks

- similar to brain activity: layer produces a signal that activates the next layer
- each layer behaves similar to a linear regression model (it consists of inputs, assigns weights and creates an output)
- if output surpasses a certain threshold, it *activates* the next layer, and becomes the input data of that layer $\rightarrow$ the activation functions allows for non-linearities
- progress/accuracy of neural network is monitored by cost function

Excursion: Neural Networks IV

A practical example

Decision to make: Should I go surfing or not?[2]

- three different factors influencing my decision

  waves good?
  beach crowded?
  shark attack?

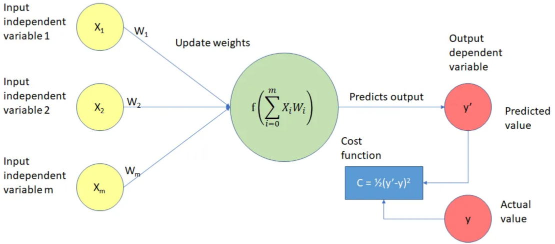
For each decision, a neural network assigns a weight which leads to an output.

[2] Source: IBM

Excursion: Neural Networks IV

A practical example II

- after each step/epoch: the output (predicted) data is compared to the actual data → loss is created: how much do the predictions of me going surfing deviate from my actual behaviour?
- goal is to minimize loss (without overfitting since the results of a neural network should apply to other data as well)



Excursion: Neural Network VI

Try out your own neural network here

Classification in R

We will use *keras* to define a deep learning model in R.

- **sequential** vs. functional models
  - in sequential models, each hidden layer is executed before another is activated
  - functional models allow to create branches which can be used to predict different outputs at the same time
- **dense** (also fully connected) vs. **convolutional** layers (for a better understanding, cf. Mandelbaum and Shalev (2016))
  - dense layers feed model with each input (token) to create an output
  - convolutional layers, not all inputs are used (but in our case, only words close to each other) → it scans through a sequence with window size *n*

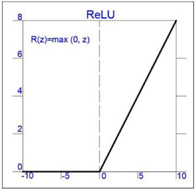
Hyperparameters I

We can adjust various parameters, some of the most important ones are

- **hidden layers**: complexity of a model, more complexity can increase precision but also lead to overfitting
- **drop-out rate**: dropping neurons to avoid overfitting (usually between 0.2-0.5)
- **learning rate**: how much weights are updated after each step
- **batches**: the chunk size which will be propagated to the neural network (the larger, the more memory needed; the smaller, the less precise)
- **number of epochs**: cycle of a learning process (from input to output and back to the input)

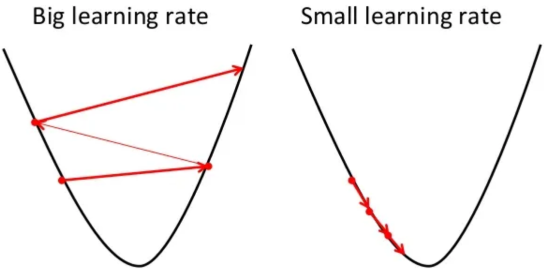
Hyperparameters II

- **activation function**: captures non-linearity of data, there are many functions (*sigmoid* often used for binary, *softmax* for multi-class predictions) as the final output layer; hidden layers commonly use *ReLu*



Every time, the threshold ($x > 0$) is surpassed, a signal is fed into the next layer

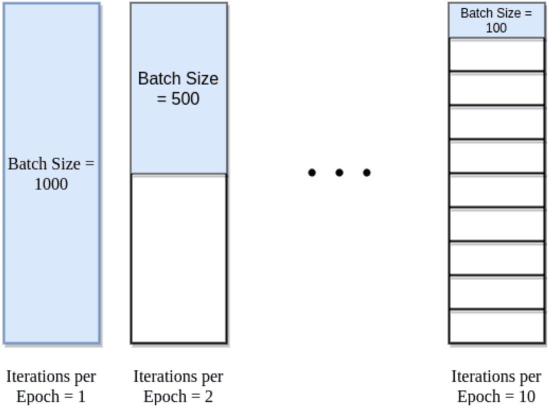
Hyperparameters III
Gradient Descent



Learning rate

How different **learning rates** affect the error

Hyperparameters IV



What **batch sizes** mean in relation to **epochs**

Hyperparameters V

What **overfitting** means in a practical example

Steps in training a deep-learning model

1. preparation of input data (train/test split, extraction of features and labels, transformation to numeric sequence structure)
2. definition of model (its input, hidden and output layer)
3. set monitoring metrics
4. training model
5. predicting test data and evaluate performance

Preparation

As before: tokenization and creation of vocabulary. In addition, **sequencing**:

```
tokenizer <- text_tokenizer(num_words = nrow(vocab)) %>%
fit_text_tokenizer(df_sub$text_prep)
sequences <- texts_to_sequences(tokenizer,
    df_sub$text_prep)
head(sequences)

# sequences must be of equal length:
max_len <- 50 # here, set to 50
features <- pad_sequences(sequences, maxlen = max_len)
```

Model definition

An example of a neural network with pre-trained embeddings as input, two hidden layers (one dense, one convolutional) and one output vector.

```
model <- keras_model_sequential() %>%
layer_embedding(
input_dim = dim(embeddings)[1],
input_length = max_len,
output_dim = dim(embeddings)[2],
weights = list(embeddings),
trainable = T) %>%
layer_conv_1d(filters = 128, kernel_size = 5,
    activation = 'relu') %>%
layer_global_max_pooling_1d() %>%
layer_dense(units = 128, activation = "relu") %>%
layer_dropout(rate = 0.4) %>%
layer_dense(units = 1, activation = "sigmoid")
```

Set monitoring metrics

In the next step, we define the loss we want to use (for binary tasks usually *binary_crossentropy*), the learning rate and the metrics shown after each epoch

```
model %>% compile(
loss = "binary_crossentropy",
optimizer = optimizer_adam(learning_rate = 0.001),
metrics = c('accuracy', metric_precision(),
    metric_recall())
)
```

Train the model

At the training stage, we add two more hyperparameters (*epochs* and *batch_size*)

```
history <- model %>% fit(
x = x_train,
y = y_train,
epochs = 20,
batch_size = 128,
validation_data = list(x_test, y_test),
callbacks = list(stop_if_no_improvement)
)
```

Prediction and evaluation

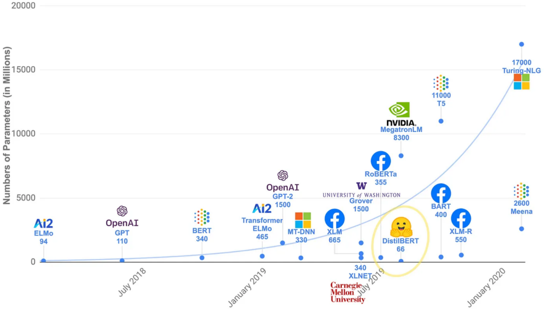
Finally, we predict the data and evaluate (e.g., with the *confusionMatrix* function of the *caret* package)

```
pred_results <- as.data.frame(predict(model, x_test))

# classify cases with probability > 0.5 as 1
pred_results$pred <- ifelse(pred_results$V1>=0.5, 1, 0)

# combine predictions with true annotated data
pred_results$true <- y_test

confusionMatrix(as.factor(pred_results$pred),
    as.factor(pred_results$true))
```

Let's do it in R

Evolution of transformer models

In parallel to the evolution of text-as-data in social science, computational science has developed more and more sophisticated models



Evolution of Transformers. Courtesy: HuggingFace

Transformer models I

Transformer models are a special type of deep learning models; they are not sequential and can pay attention to multiple inputs at the same time

- **attention layer:** features to pay particular attention to
- e.g., in a translation task, the transformer can pay attention to surrounding words which are important for the correct translation (e.g., 'I am' → 'Je suis')

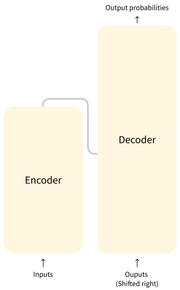
Vaswani et al. (2017) → they can be parallelized and are much faster!

Transformer models II

Transformer consider not only one layer of information but build several layers that better explain what a particular **feature** means in a **context**.

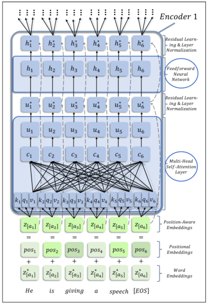
- semantical (linguistic meaning)
- morphological (form of a word)
- syntactic (sentence structure)

How transformer models work I



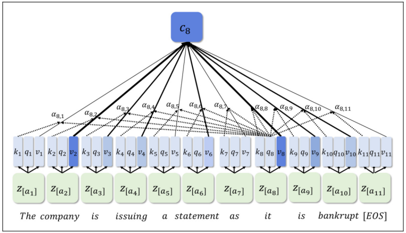
Encoder translates words into multidimensional representations, decoder translates representations into words.

How transformer models work II



Input tokens are embedded, get a positional encoding and are fed into a multi-head attention layer. This creates context-specific embeddings for each input token. (Wankmüller 2022, p. 24)

How transformer models work III



An example of an attention layer: transformers focus on the context and detect the syntactic relationship between the terms *company* and *it* (Wankmüller 2022, p. 25).

How transformer models work IV

A few more remarks

- There are not one but eight (!) different attention layers, each creating individual context vectors (e.g., capturing semantic, syntactic, morphological structure).
- A decoder works in a very similar way (just in reverse) with one exception: it is **autoregressive**: subsequent words are masked so it has to predict them by itself

Different models, different architectures

- **encoder-decoder** architectures are usually used for translation of text
- for text classification, we use **encoder-only models** (e.g. *BERT*, *RoBERTa*)
- **decoder-only** is used for generating texts

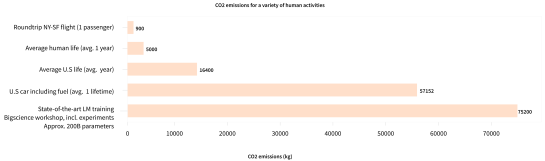
Pre-training

In pre-training of natural language processing, we need a large corpus that is a very general representation of text.

- usually pre-training is done on large databases (like on Stanford Sentiment Treebank or on Tweets)
- many of these tasks are self-supervised; for instance, by masking a token in a sentence and predicting it or by predicting the order of two sentences (that's the pre-training of BERT)
- some more complex pre-training tasks involve training data (like translation)

Pre-training

Pre-training is computationally expensive (and not that good for the environment!)



Fine-tuning I

Luckily, most of us won't care about training a transformer models. We will usually rely on pre-trained models which can be downloaded for different applications from Hugging Face → for free!

There are **two options on how to proceed**

1. feature extraction: just use the pre-trained model as is (not domain-specific)
2. **fine-tune** the pre-trained models; this adjusts the parameters to our domain

cf. Wankmüller (2022)

Fine-tuning

- fine-tuning allows transfer learning: we take a general model and make it domain-specific
- during adaptation, the weights of our pre-trained models can be adjusted
- in neural network lingo, we add another output layer to our model $\rightarrow$ usually, the rest of the model architecture remains the same

Application in Python

Thanks to the *transformers* module, we do not need to apply all of these steps by hand.

transformers:

- pre-processes data (tokenizer)
- applies a task (classification, translation, etc.)
- returns the results of the tasks

Let's switch to Colab.

A few notes of caution

These models allow to perform tasks we could not have done before, **but**...

- ...they are extremely data intense and need massive computing power (in an era of climate crisis!)
- ...they replicate biases
- ...their architecture is complex: what happens under the hood is very difficult to interpret

Resources

Useful links

- **Tutorial on deep learning**: Practical Deep Learning
- **Tutorial on transformers in Colab** A Practical Introduction to Transformers by Moritz Laurer
- Models for Hugging Face
- GloVe Documentation

Thank you for your attention!

...and thanks to **Theresa Gessler** and some inspiration from her CTA workshop

...and **Moritz Laurer** and his Workshop on Transformer at COMPTEXT

References I

Kusner, M. J., Sun, Y., Kolkin, N. I., & Weinberger, K. Q. (2015). From Word Embeddings To Document Distances.

Mandelbaum, A., & Shalev, A. (2016). Word Embeddings and Their Use In Sentence Classification Tasks. https://doi.org/10.48550/arXiv.1610.08229

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 1532–1543.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. https://arxiv.org/abs/1802.05365

Rheault, L., & Cochrane, C. (2020). Word Embeddings for the Analysis of Ideological Placement in Parliamentary Corpora. *Political Analysis*, *28*(1), 112–133. https://doi.org/10.1017/pan.2019.26

References II

Rodriguez, P., & Spirling, A. (2021). Word Embeddings: What works, what doesn't, and how to tell the difference for applied research. *The Journal of Politics*. https://doi.org/10.1086/715162

Rodriguez, P. L., Spirling, A., & Stewart, B. M. (2023). Embedding Regression: Models for Context-Specific Description and Inference. *American Political Science Review*, *117*(4), 1255–1274. https://doi.org/10.1017/S0003055422001228

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ukasz Kaiser, Ł., & Polosukhin, I. (2017). Attention is All you Need. *Advances in Neural Information Processing Systems*, *30*.

Wankmüller, S. (2022). Introduction to Neural Transfer Learning With Transformers for Social Science Text Analysis. *Sociological Methods & Research*, 1–77. https://doi.org/10.1177/00491241221134527