Text Mining and Natural Language Processing

2023-2024

Alessandro Ghiotto 513944

SelectWise

Introduction

Each item consists of a question, 8 multiple choices (from 'A' to 'H') and two facts that provide information about the question, the task being to select the correct answer. The multiple-choice task can be seen as a single-label, multi-class classification between the 8 possible alternatives. The difficulty in seeing the multiple choice task as a classification is that the 8 classes {'A', 'B',..., 'H'} (or {'0', '1',..., '7'}) don't have a fixed meaning, but the meaning of each class is given by the choices in the sample.

Data

I chose the QASC dataset from AllenAI, https://huggingface.co/datasets/allenai/qasc. I have simplified the dataset task a bit, as in the original task the facts have to be retrieved from a text corpus, instead I use them as if they were given. The questions are about grade school science. Each sample have this form:

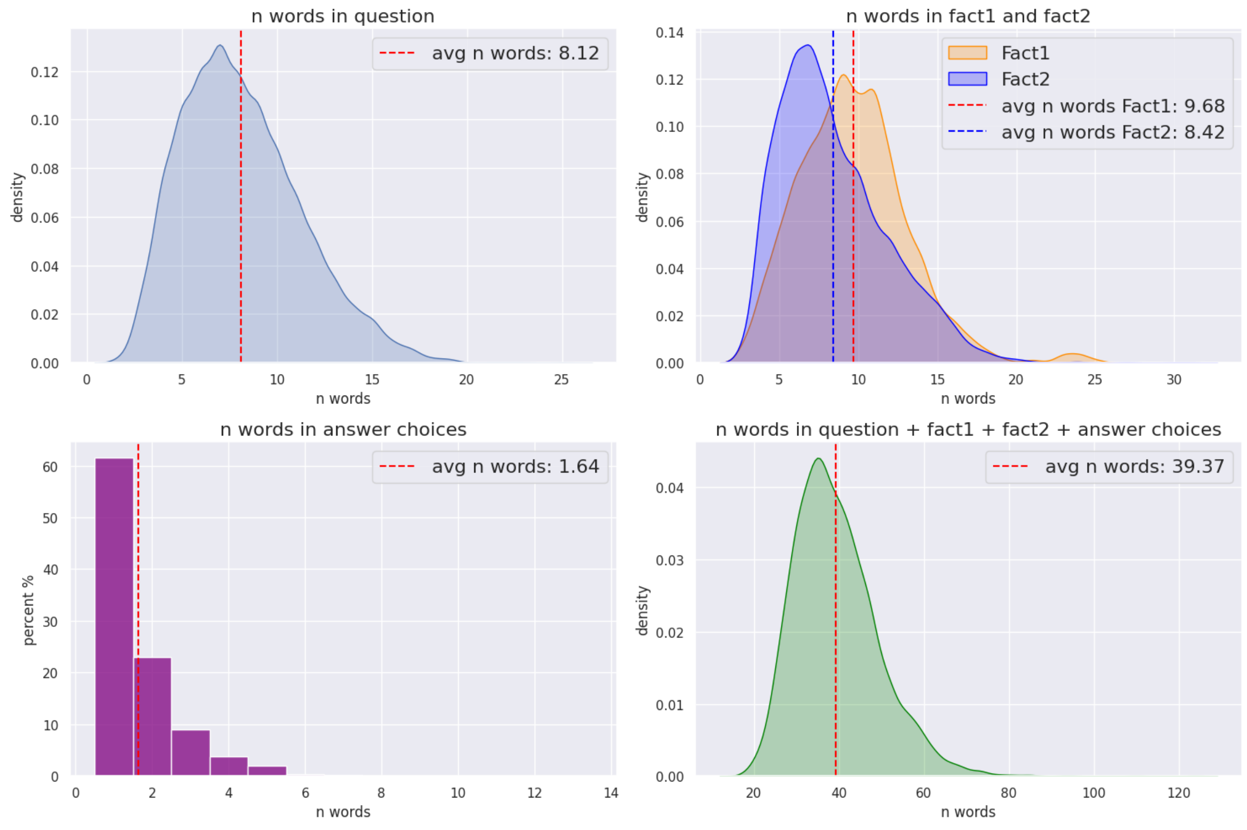
"**answerKey**": "F",
“**choices**": {"label": ["A", "B", "C", "D", "E", "F", "G", "H"],
"text": "sand", "occurs over a wide range", "forests", "Global warming", "rapid changes occur", "local weather conditions", "measure of motion", "city life"]},
"combinedfact": "Climate is generally described in terms of local weather conditions",

"**fact1**": "Climate is generally described in terms of temperature and moisture.",
"**fact2**": "Fire behavior is driven by local weather conditions such as winds, temperature and moisture.",
"formatted_question": "Climate is generally described in terms of what? (A) sand (B) occurs over a wide range (C) forests (D) Global warming (E) rapid changes occur (F) local weather conditions (G) measure of motion (H) city life",
"id": "3NGI5ARFTT4HNGVWXAMLNBMFA0U1PG",
"**question**": "Climate is generally described in terms of what?"

The main fields are: question, fact1, fact2, choices (which contains the height possible alternatives) and answerKey (the label). Here I have the number of samples and the number of words in train, val (dev) and test dataset. In the number of words I have counted all the words in the following fields: *question, fact1, fact2, choices['text']*

	Train	Val	Test
Number of samples	7323	811	926
Total number of words	288301	31779	35963

Here we can see some more detailed statistics about the single fields of our items, computed on the train data only.



In particular we have the distribution of the number of words in the *question* (*avg=8.12*), in *fact1* (*avg=9.68*) and *fact2* (*avg=8.42*), in each of the possible *choices* (*avg=1.64*) and in the whole item (*avg=39.37*). In the following bar plot instead we can see that the eight classes/choices are balanced.

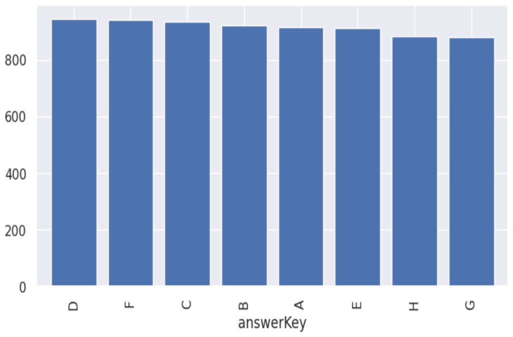
Here we have some other statistics about the choices:

- number of choices costituted by a single word: 36042

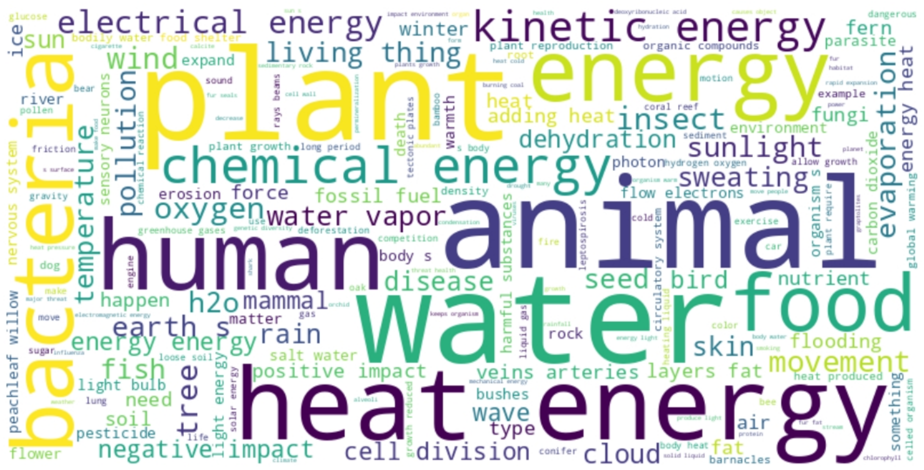
- number of choices costituted by more than one word: 22542

- percentage of choices costituted by a single word: 61.522 %

- max number of words of a choice: 13



In the word cloud we can easily recognize that the questions are about grade school science.



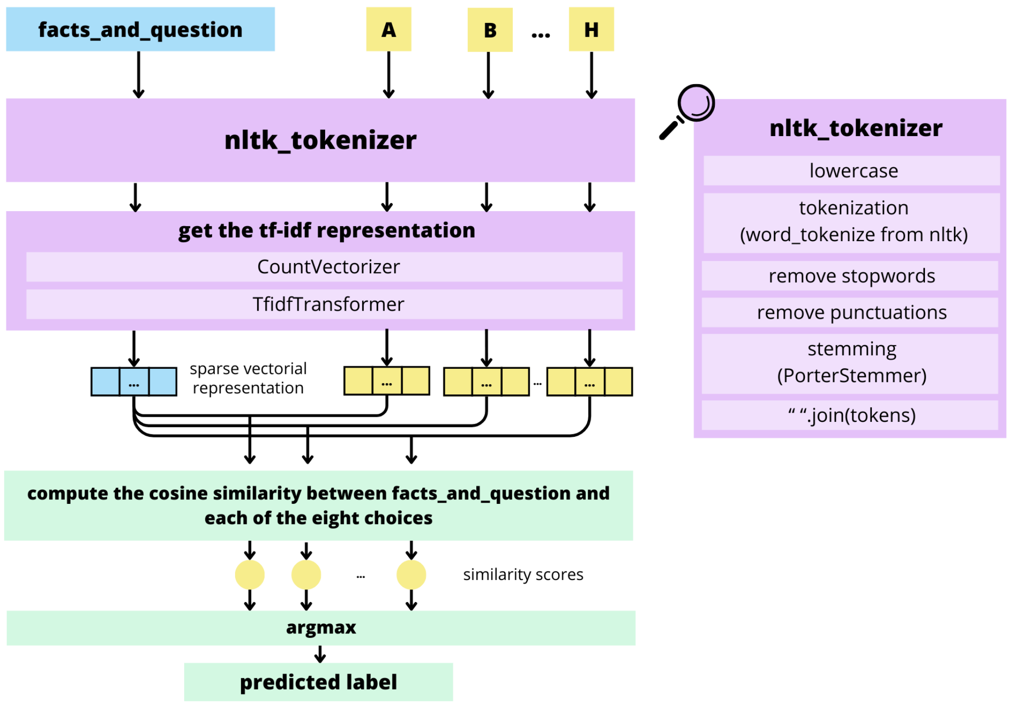
Methodology

Notebook 1 – Count based methods

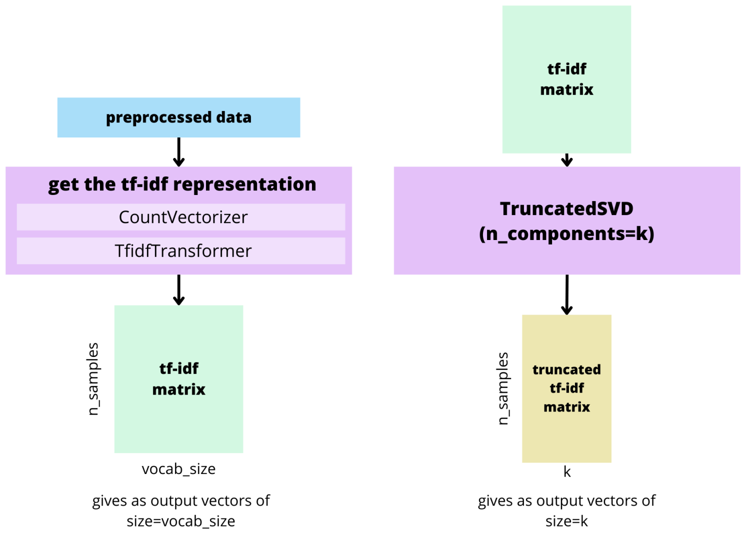
In the first notebook I have started with some simple count based method, which are the tf-idf and the n-gram language models.

Representing documents with TF-IDF weighting

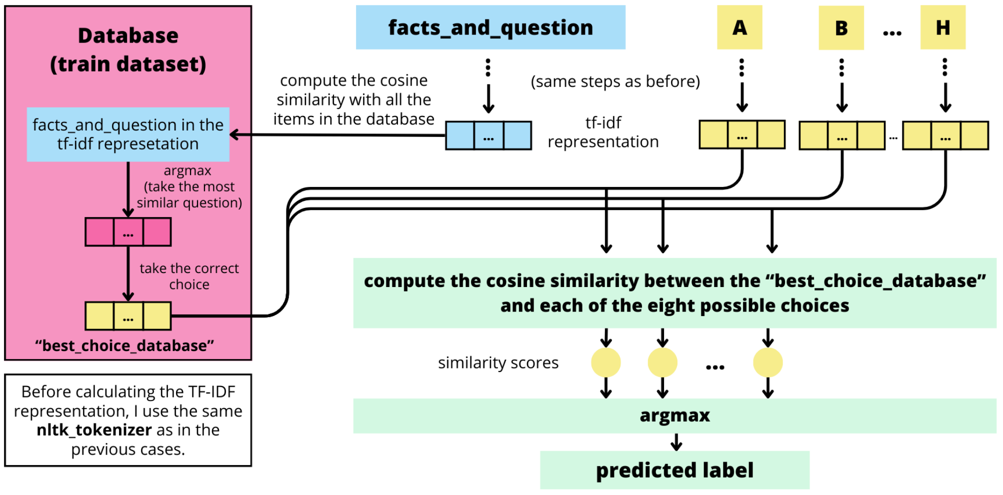
I took the '*facts_and_question*' column (result of *df['facts_and_question'] = df['fact1'] + ' ' + df['fact2'] + ' ' + df['question']*) and each of the possible choices, then applied the *nltk_tokenizer*, got the **tf-idf representation** and finally took as predicted label the choice that had the **highest cosine similarity** to *'facts_and_question'*. With the *CountVectorizer* and the *TfidfTransformer* (from sklearn) I get the tf-idf matrix, which is used to get the tf-idf representation of each text string. In the image below we can see how to get the output label from one item. First I compute the tf-idf matrix, and then I use it for each sample in the pipeline below.



Then I have tried to use **TruncatedSVD** for obtaining a dense representation of the tf-idf matrix, the model is identical to the one in the figure above, we simply change the tf-idf matrix with the truncated one. I have tried with *n_components* = 200, 50 and 10

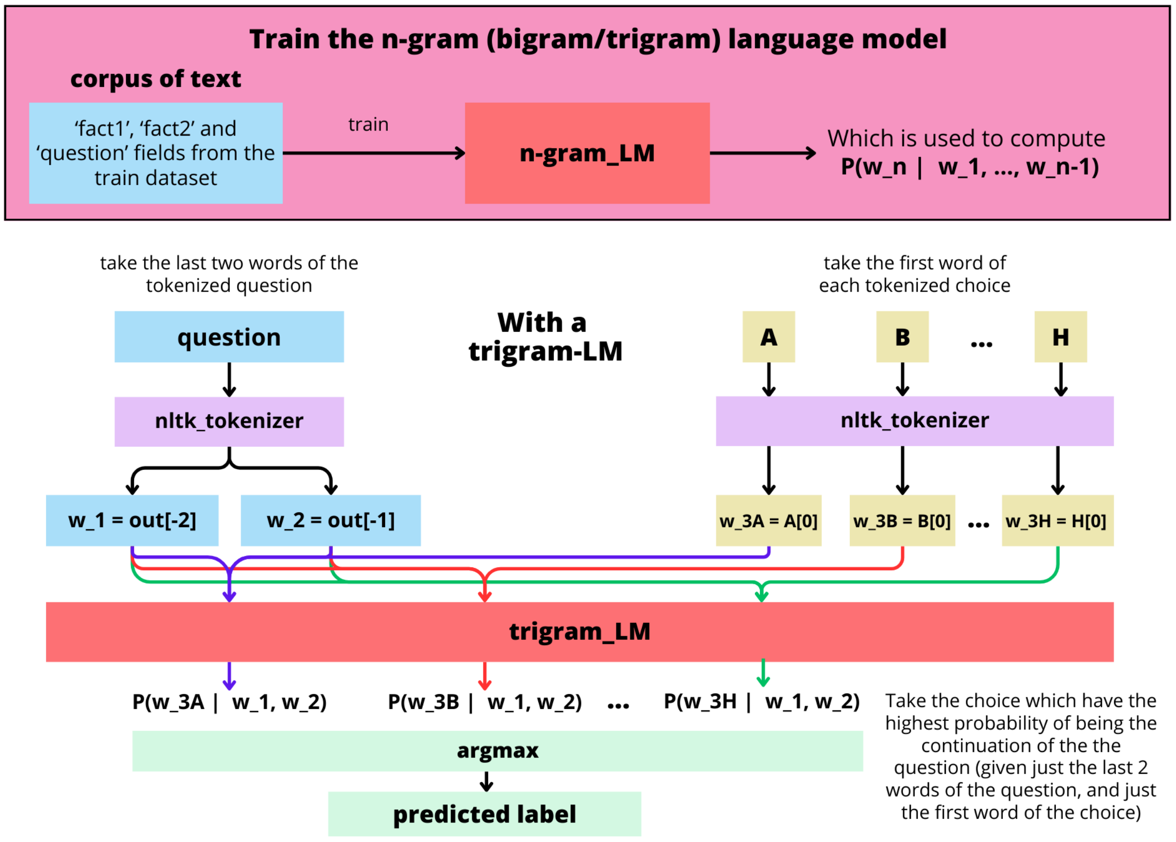

preprocessed data
get the tf-idf representation
CountVectorizer
TfidfTransformer
n_samples
tf-idf matrix
vocab_size
gives as output vectors of size=vocab_size
tf-idf matrix
TruncatedSVD (n_components=k)
n_samples
truncated tf-idf matrix
k
gives as output vectors of size=k

As a final method with tf-idf, I tried to retrieve the correct answer from the samples in the training data. For each sample, I took the **sample from the train with the highest cosine similarity** computed on the '*facts_and_question*' entry (still on the tf-idf representation, with the same preprocessing). Then I took the answer among the possible ones that was most similar to the correct answer (the *'best_choice_database'* from the image below) of the retrieved sample.


Database
(train dataset)
facts_and_question in the tf-idf represetation
argmax (take the most similar question)
take the correct choice
"best_choice_database"
Before calculating the TF-IDF representation, I use the same nltk_tokenizer as in the previous cases.
facts_and_question
compute the cosine similarity with all the items in the database
(same steps as before)
tf-idf representation
A
B
...
H
compute the cosine similarity between the "best_choice_database" and each of the eight possible choices
similarity scores
argmax
predicted label

N-gram LM based classification

I have trained an n-gram LM (a bigram and a trigram) on the '*fact1*', '*fact2*' and '*question*' columns of the train dataset. Then asked to the model which of the eights choices was the most probable for continuing the question. For the choices I have taken just the first word and for the question just the last words (one for the bigram, two for the trigram). The nltk_tokenizer is the same one used with the tf-idf method.


Train the n-gram (bigram/trigram) language model
corpus of text
'fact1', 'fact2' and 'question' fields from the train dataset
train
n-gram_LM
Which is used to compute P(w_n | w_1, ..., w_n-1)
take the last two words of the tokenized question
With a trigram-LM
take the first word of each tokenized choice
question
A
B
...
H
nltk_tokenizer
nltk_tokenizer
w_1 = out[-2]
w_2 = out[-1]
w_3A = A[0]
w_3B = B[0]
...
w_3H = H[0]
trigram_LM
P(w_3A | w_1, w_2)
P(w_3B | w_1, w_2)
...
P(w_3H | w_1, w_2)
Take the choice which have the highest probability of being the continuation of the the question (given just the last 2 words of the question, and just the first word of the choice)
argmax
predicted label

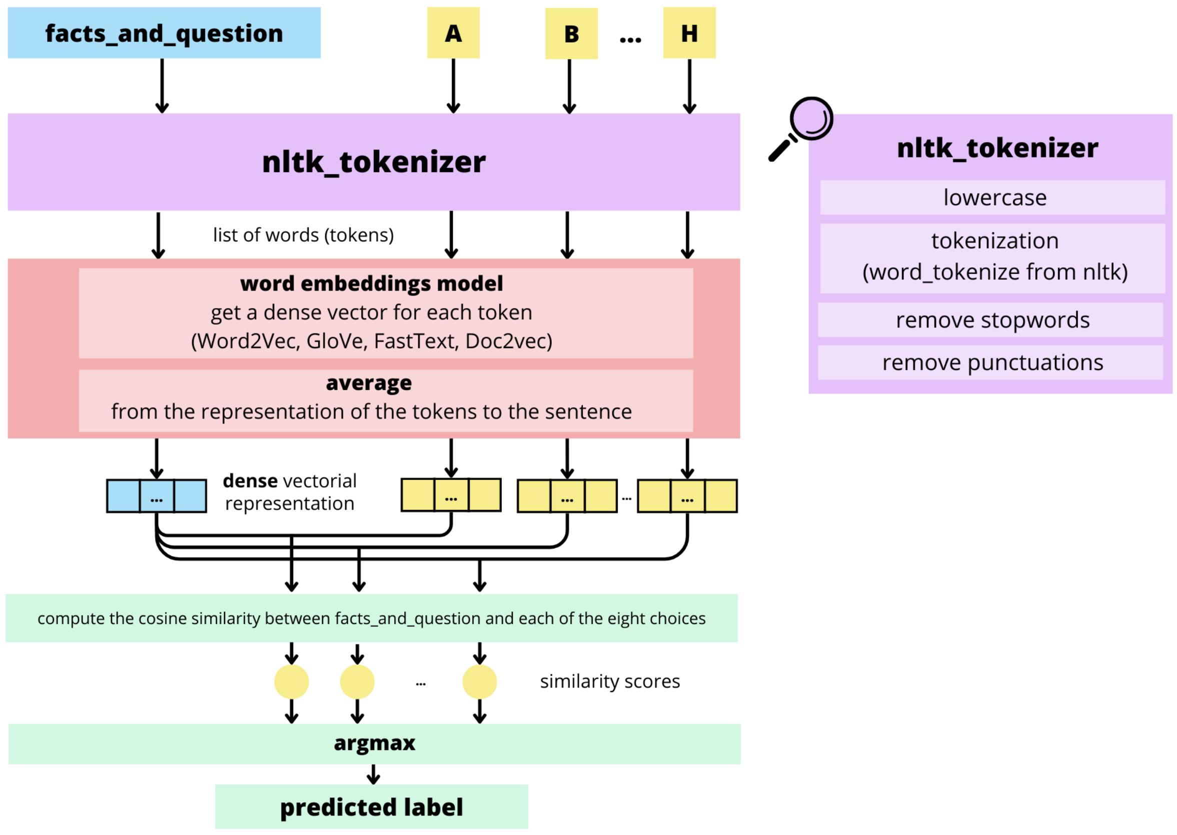
Notebook 2 – Word Embeddings

Representation by means of static word embeddings

For the representation of the words I use the **dense vectors** obtained by models like Word2Vec, as for the representation of the sentence I simply took the average. As with the tf-idf, I used the '*facts_and_question*' column and took the choice with the **highest cosine similarity** to this column. The method is exactly the same as with tf-idf, just the representation changes. I always used **window=5**. I have tried the following models:

- **Word2Vec**: trained on train data with k=300 and k=100 with the skip-gram architecture, and the pretrained '*word2vec-google-news-300*'
- **GloVe**: '*glove-wiki-gigaword-100*' and '*glove-wiki-gigaword-300*'
- **FastText**: trained on train data with k=300 and k=100 with the skip-gram architecture, and the pretrained '*fasttext-wiki-news-subwords-300*'
- **Doc2Vec**: trained on train data with k=300 (with DistributedMemory and DistributedBoW) and k=100 (with DistributedBoW)

First we train the word embeddingd model on the '*fact1'*, '*fact2'* and '*question'* columns of the train dataset (which undergoes the same nltk_tokenizer seen below). Then we take the word embedding matrix from the model and use it for obtaining the dense vector, like in the following picture.


facts_and_question
A
B
...
H
nltk_tokenizer
list of words (tokens)
word embeddings model
get a dense vector for each token
(Word2Vec, GloVe, FastText, Doc2vec)
average
from the representation of the tokens to the sentence
dense vectorial representation
compute the cosine similarity between facts_and_question and each of the eight choices
similarity scores
argmax
predicted label
nltk_tokenizer
lowercase
tokenization
(word_tokenize from nltk)
remove stopwords
remove punctuations

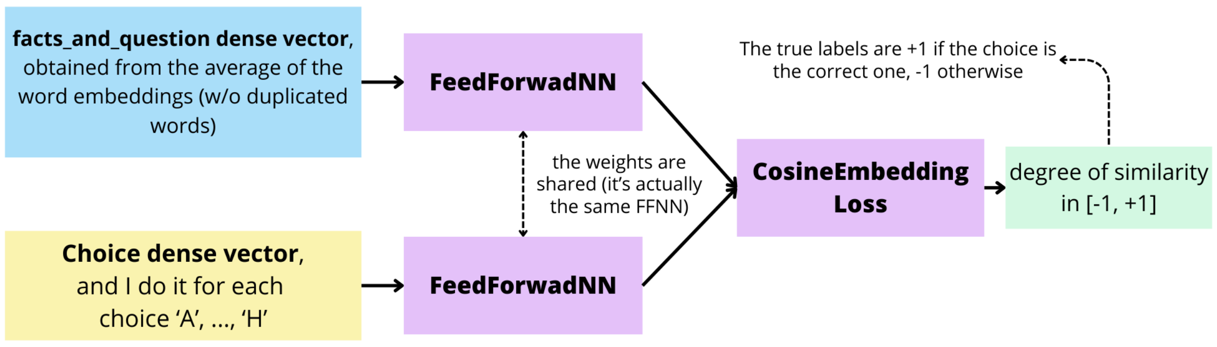
Other ways of combining word embeddings

Before, to get the representation of the sentence, I took the average of the word embeddings. But then I also tried to **remove the duplicated words** from the sentence, I just see if a word is present or not (and then take the average). And I also tried to take a **weighted average** of the word embeddings, using the **IDF** scores as the weight for each word. Finally, I tried to combine the two methods.

Other ways of choosing the answer

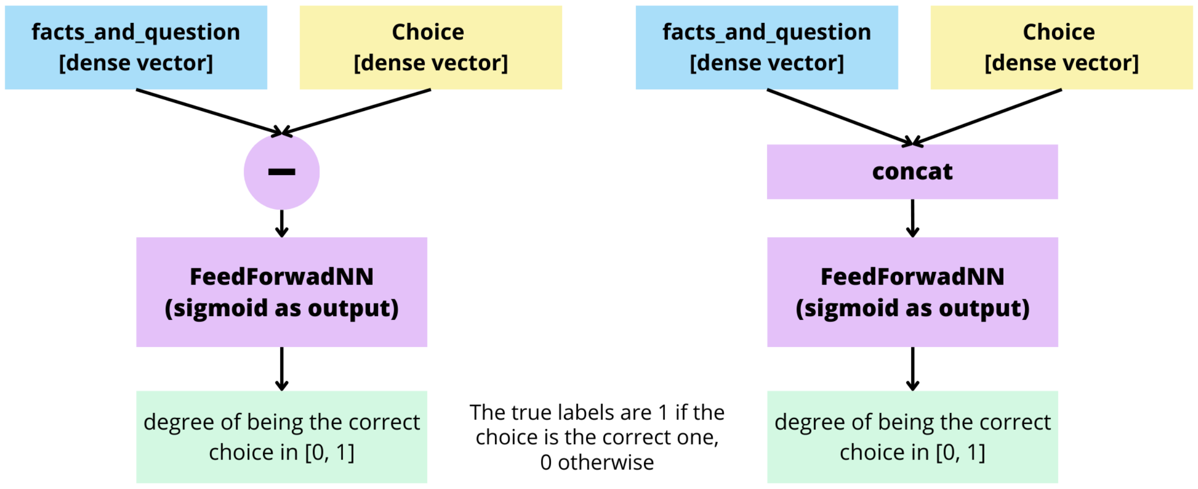
In the first method, I chose the answer with the highest cosine similarity to 'facts_and_question', but I can train a neural network that tells me whether a choice is correct or not, given the representation of 'facts_and_question' and each choice.

SiameseNN: the two vectors (question and choice) go through the same FFNN, then we use the **CosineEmbeddingLoss** on the two outputs. We train the model to produce outputs with higher/lower cosine similarity when the choice is right/wrong.



facts_and_question dense vector, obtained from the average of the word embeddings (w/o duplicated words)
FeedForwadNN
The true labels are +1 if the choice is the correct one, -1 otherwise
the weights are shared (it's actually the same FFNN)
CosineEmbedding Loss
degree of similarity in [-1, +1]
Choice dense vector, and I do it for each choice 'A', ..., 'H'
FeedForwadNN

FeedForwardNN: Instead of processing the question and the choice alone and then computing the difference (as in the SiameseNN), I combine the choice and the question and then run the result through a FFNN. To combine the output I have tried to take the **difference** or **concatenate** the two vectors. As output I have a sigmoid function, and the target labels are 0 if the choice is wrong / 1 if the choice is correct.



facts_and_question [dense vector]
Choice [dense vector]
−
FeedForwadNN (sigmoid as output)
degree of being the correct choice in [0, 1]
The true labels are 1 if the choice is the correct one, 0 otherwise
facts_and_question [dense vector]
Choice [dense vector]
concat
FeedForwadNN (sigmoid as output)
degree of being the correct choice in [0, 1]

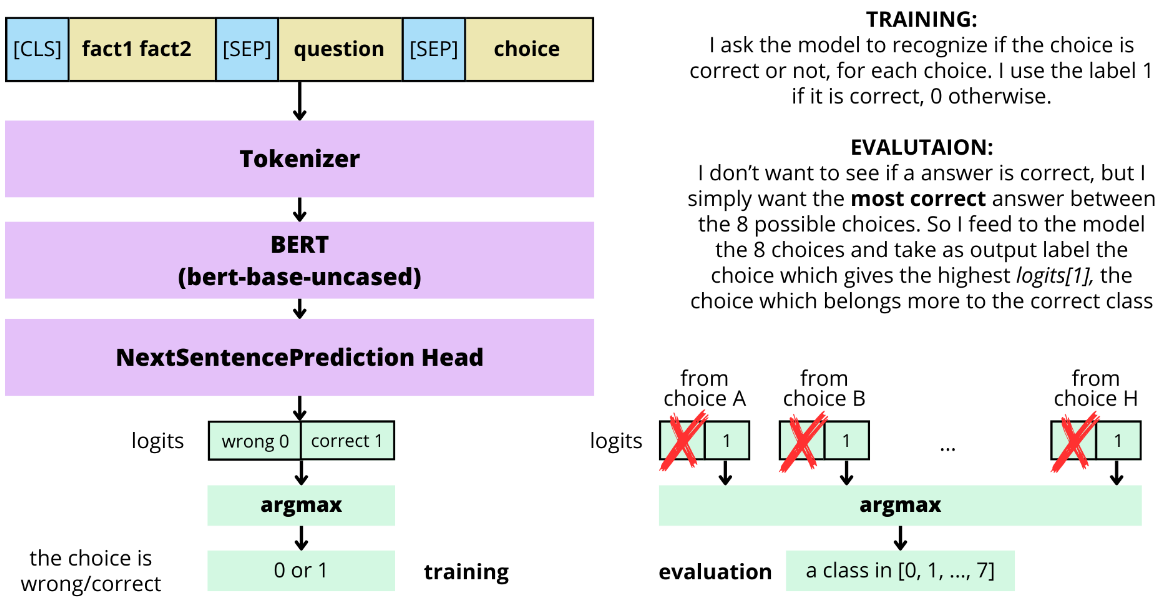
In both cases (siameseNN, FFNN), during training I ask if the choice is correct or not, but during evaluation I consider all 8 choices of a sample simultaneously and take the most correct one (argmax of the eights outputs given by the eights possible choices).

Notebook 3 – Transformer Encoder Only – BERT

I have 8 inputs for each sample, of this type:
"[CLS] fact_1 fact_2 [SEP] question [SEP] choice_i [SEP]" for each choice i in ['A','B',…,'H'].
BERT (in particular '*bert-base-uncased*') encodes the input sentence, then the output of the **[CLS]** token is fed to a classification head. Each token of the input sequence has as output a vector, which can be seen as a contextualized embedding of the input token.

Binary classification

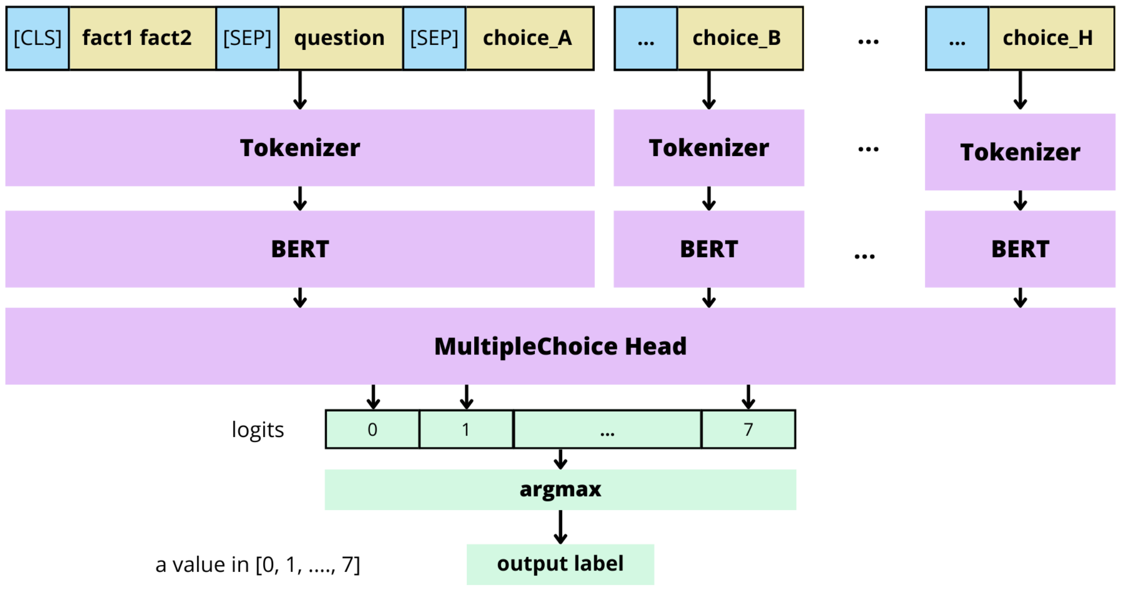
I train the model like a **NextSentencePrediction** task, I simply ask if this choice is correct or not. So as output I have just two values, one associated to the positive class and the other to the negative one.


[CLS]
fact1 fact2
[SEP]
question
[SEP]
choice
Tokenizer
BERT (bert-base-uncased)
NextSentencePrediction Head
logits
wrong 0
correct 1
argmax
the choice is wrong/correct
0 or 1
training
TRAINING:
I ask the model to recognize if the choice is correct or not, for each choice. I use the label 1 if it is correct, 0 otherwise.
EVALUTAION:
I don't want to see if a answer is correct, but I simply want the most correct answer between the 8 possible choices. So I feed to the model the 8 choices and take as output label the choice which gives the highest logits[1], the choice which belongs more to the correct class
from choice A
from choice B
from choice H
logits
1
1
...
1
argmax
evaluation
a class in [0, 1, ..., 7]

As in the previous notebook (the part with the SiameseNN and the FeedForwardNN), during train I ask if it is wrong or correct, on evaluation I take the most correct choice (argmax).

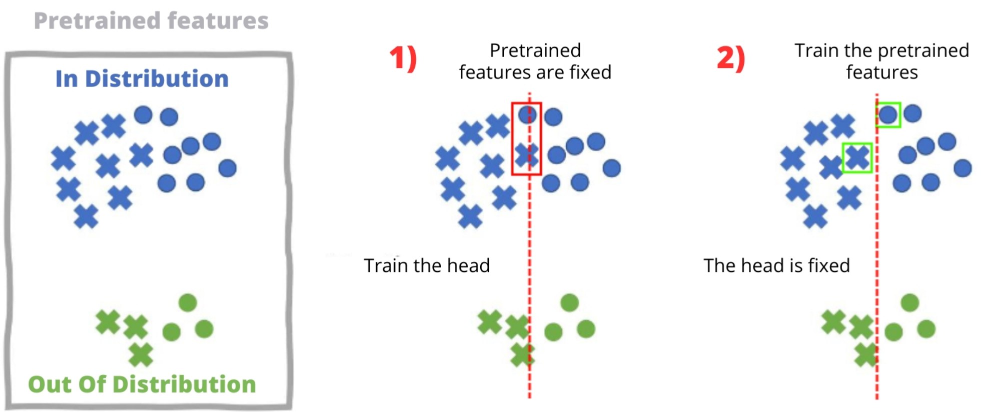
Multiclass classification

I train the model like a **single-label multi-class classification** task, for each choice I have an output, the one which is higher is the correct choice (given by the model). Now I have as output 8 values, I take the argmax, and gives me a label in {0, 1, 2, 3, 4, 5, 6, 7}.



Different ways of tuning a pretrained model

I have trained the model with **fine-tuning**, **linear probing** and with the **combined method'**. In this last method I first train the classification head (while the pretrained features are fixed), then I train the features (while the head is fixed). This method should get the best of both linear probing and fine-tuning, the idea being that it should be better than fine tuning on OOD samples because we train the classification head first while the features are fixed, this should result in a head that is less conditioned by the training data. And should be better of linear probing since we also train the pretrained features.



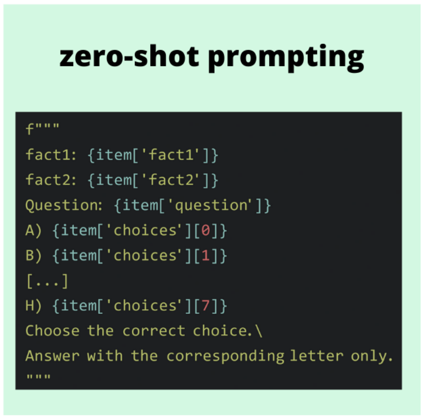
Notebook 4 – LLM prompting

As LLM I have chosen **'DeciLM-7B-instruct'**, a model for short-form instruction following, https://huggingface.co/Deci/DeciLM-7B-instruct. I follow the specific prompt for this LM:

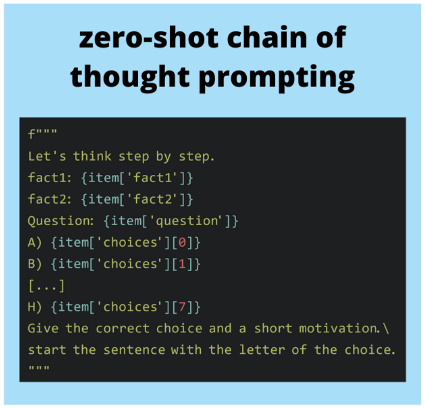
```
### System:
You are an AI assistant that follows instruction extremely well. Help as much as you can.
### User:
{propt}
```

All the values used are simple strings of text, exactly as given by the original dataset. In the 'choices' key we have a list of eight strings, which are the eight possible choices, the first element corresponds to the choice A, the second to B,… and so on until the letter H. Then I take the answer to the prompt given by the LLM and in particular I **take the first letter**, which should be a value in [A, B, …, H]. Otherwise, if it's not one of these letters, I skip the sample. I have tried the following prompting strategies:

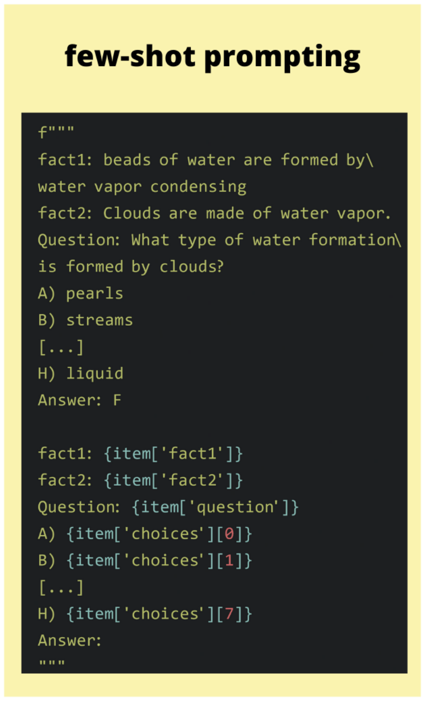
- Zero-Shot Prompting
- Zero-Shot Chain of Thought Prompting
- Few-Shot Prompting
- RAG inspired Few-Shot Prompting: instead of giving as context one example at random, I choose the one which is the most similar (between all the samples in the train dataset) to the current question. The cosine similarity is computed between the questions in the tf-idf representation.



```
f"""
fact1: {item['fact1']}
fact2: {item['fact2']}
Question: {item['question']}
A) {item['choices'][0]}
B) {item['choices'][1]}
[...]
H) {item['choices'][7]}
Choose the correct choice.\
Answer with the corresponding letter only.
"""
```



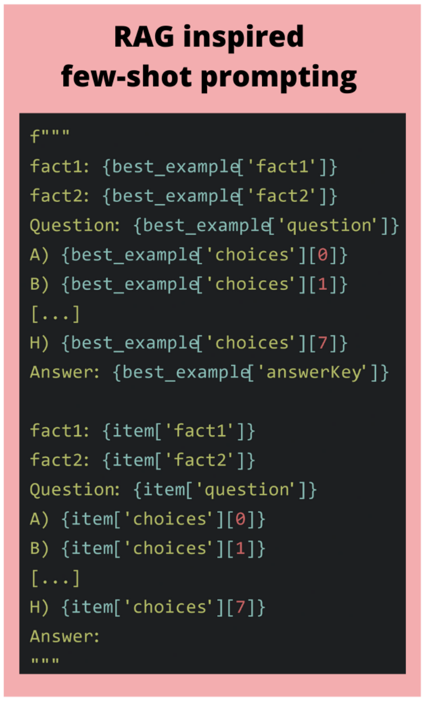
```
f"""
Let's think step by step.
fact1: {item['fact1']}
fact2: {item['fact2']}
Question: {item['question']}
A) {item['choices'][0]}
B) {item['choices'][1]}
[...]
H) {item['choices'][7]}
Give the correct choice and a short motivation.\
start the sentence with the letter of the choice.
"""
```



```
f"""
fact1: beads of water are formed by\
water vapor condensing
fact2: Clouds are made of water vapor.
Question: What type of water formation\
is formed by clouds?
A) pearls
B) streams
[...]
H) liquid
Answer: F

fact1: {item['fact1']}
fact2: {item['fact2']}
Question: {item['question']}
A) {item['choices'][0]}
B) {item['choices'][1]}
[...]
H) {item['choices'][7]}
Answer:
"""
```

The first example (context) is given by the first sample in the train dataset, **dataset_train[0]**



```
f"""
fact1: {best_example['fact1']}
fact2: {best_example['fact2']}
Question: {best_example['question']}
A) {best_example['choices'][0]}
B) {best_example['choices'][1]}
[...]
H) {best_example['choices'][7]}
Answer: {best_example['answerKey']}

fact1: {item['fact1']}
fact2: {item['fact2']}
Question: {item['question']}
A) {item['choices'][0]}
B) {item['choices'][1]}
[...]
H) {item['choices'][7]}
Answer:
"""
```

The **best_example** is the sample that has the highest cosine similarity, computed on the TF-IDF representation of the question, with the question of the current item.

Notebook 5 – Evaluation on the Test Set

Here I look at the performances on the test dataset of the two best methods (based on the accuracy on the validation dataset): BERT (tuned with 'combined_method') and DeciLM (with few-shot prompting strategy).

Result and Analysis

Here I will look at the results. I only care about the **accuracy on the validation dataset**. I have also saw the F1 score in the notebooks, but since the classes don't have any real meaning and are also balanced in the end the F1 score was always very close to the accuracy. Also with these labels which have no a specific meaning, we don't care about precision and recall, we have to make the right choice and that's it. In the table we have (from the left to the right) the approach used (below the notebook title), the accuracy on the train dataset and the accuracy on the validation dataset (in the format 0.00%). I also put the accuracy on the train dataset, just to see if a model is **overfitted** or not.

Notebook 1 – Count based methods	**Train acc %**	**Val acc %**
Normal tf-idf	88.08%	**86.68%**
tf-idf truncated with SVD – n_components=200	78.81%	83.48%
tf-idf truncated with SVD – n_components=50	63.50%	72.75%
tf-idf truncated with SVD – n_components=10	43.18%	52.04%
tf-idf retrieval from train set	30.93%	20.59%
Trigram LM based classification	16.97%	15.41%
Bigram LM based classification	21.66%	13.93%

The **'*normal tf-idf*'** approach is the most successful, truncating the tf-idf matrix didn't increase the result, but losing just the 3% of accuracy by reducing the dimensionality of the representation from *vocab_size – 200 = 6267* is a very good compromise. The last three methods probably requires more data, and an n-gram LM has too short a context window for the task. As the topics dealt with in the questions are slightly different between training and validation, we can see a large gap between training and validation accuracy in *'tf-idf retrieval from train set'* and *'bigram LM based classification methods'*. I think the result obtained with tf-idf is very good, considering how simple it is as a method, but this can be given by the simplicity of the questions, since tf-idf just see the words occurrences, doesn't extracts any relation between the words.

Notebook 2 – Word Embeddings	**Train acc %**	**Val acc %**
Word2Vec model trained on dataset – k=300	74.31%	55.98%
Word2Vec model trained on dataset – k=100	73.99%	55.73%

Word2Vec pretrained – word2vec-google-news-300	63.85%	**64.49%**
GloVe pretrained – glove-wiki-gigaword-100	39.71%	37.98%
GloVe pretrained – glove-wiki-gigaword-300	53.09%	53.02%
FastText model trained on dataset – k=300	70.76%	57.46%
FastText model trained on dataset – k=100	70.76%	58.08%
FastText pretrained – fasttext-wiki-news-subwords-300	52.30%	49.94%
Doc2Vec model trained on dataset – k=300 – DM	55.93%	42.17%
Doc2Vec model trained on dataset – k=300 – DBoW	72.70%	61.41%
Doc2Vec model trained on dataset – k=100 – DBoW	74.30%	63.13%
word2vec-google-news-300 – remove duplicated words	77.24%	**78.05%**
word2vec-google-news-300 – words weighted by their IDF score	67.08%	69.67%
word2vec-google-news-300 – words weighted by their IDF score – remove duplicated words	78.19%	**79.90%**
word2vec-google-news-300 – SiameseNN	47.17%	45.99%
word2vec-google-news-300 – FeedForwardNN – difference	90.78%	**76.33%**
word2vec-google-news-300 – FeedForwardNN – concatenate	86.88%	67.82%

The first 12 methods use the average of the word embeddings as a representation of the sentence. The last 6 methods use the pretrained *'word2vec-google-news-300'* model as an embedding. The last 3 methods remove the duplicated words from the sentence. All models trained on our dataset (Word2Vec, FastText and Doc2Vec) gave very similar results, the highest being Doc2Vec (k=100, DboW) with val_acc=63.13. It's interesting looking at the fact that **Doc2Vec with k=100** got the highest result on validation, not just between all the Doc2Vec model, but between all the models trained on our dataset. It's like the sentences are just simple, so having more dimension to fill which the model doesn't need it just create noise in the representation, and maybe this is the downside in using the argmax of the cosine similarity, that the features can't be ignored (instead in Neural Networks we can have very low weights for specific values).

The best overall on unseen data is the pretrained ***'word2vec-google-news-300'***. Removing the duplicate words gives an incredible increase in accuracy. The neural methods on the output don't give a very good result, taking the highest cosine similarity between the choice and the question is simple and works just fine. It's very interesting how just **removing the duplicated words** in the sentence increased accuracy so much.

Notebook 3 – Transformer Encoder Only	**Train acc %**	**Val acc %**
BERT – binary classification – fine-tuning	97.87%	97.41%
BERT – multiclass classification – fine-tuning	97.94%	97.90%
BERT – multiclass classification – linear probing	99.81%	97.90%
BERT – multiclass classification – combined method	99.89%	**98.15%**

All models use the ***'bert-base-uncased'*** version of BERT. As expected, all the results are very good, because the **contextualised embeddings** given by a transformer encoder-only architecture can capture much more than simple static embeddings, they have a much better chance of learning the relationships between the tokens in the sequence. First I tried both models for binary and multiclass classification, then the other training methods are done only with the multiclass architecture, because the results are equivalent, but this last one is easier and more convenient to use. The combined training method achieved the highest accuracy at validation, but in the end the difference is very small.

Notebook 4 – LLM Prompting	**Val acc %**
DeciLM – zero-shot prompting	96.18%
DeciLM – zero-shot chain of thought prompting	96.63%
DeciLM – few-shot prompting	**97.41%**
DeciLM – RAG inspired few-shot prompting	96.42%

All models use the ***'DeciLM-7B-instruct'*** LLM. I have tried different prompts for each category, just changing the way the sentence is phrased or for example the order of the elements in the prompt, but in the table above I have just put the highest accuracy I got. I only tested the model on the validation dataset, as testing the training dataset is just a waste of resources, as this LLM was not fine-tuned for our task. We can't evaluate overfitting, it's just another unseen dataset like the validation dataset, with the difference that we can't use the accuracy of the train for comparisons with the other models. The best result is obtained by the **few-shot prompting**.

The zero-shot chain of thought prompting took more than ten times longer than the zero-shot prompting, because in the zero-shot chain of thought we ask the model to generate more text, and because the text has to be generated sequentially, it takes a lot of time. There isn't much difference between the zero-shot and the few-shot (in terms of time required), the few-shot is just a little higher because the model takes a longer string of text as input, but the difference is almost meaningless. I like the idea of few-shot prompting a lot, because we don't have to specify what to do with a sentence, we just give a pattern.

Notebook 5 – Evaluation on the Test	**Test acc %**	**Time elapsed (s) / n_samples**
DeciLM – few-shot prompting	**99.03%**	2.23E-01
BERT – multiclass classification – combined method	97.19%	**6.23E-03**

Finally, I looked at the performance of the best models on the test dataset. I have also given the average time taken by the model to predict the result for a sample (total elapsed time in seconds divided by the number of samples) to highlight the fact that an **LLM takes much longer**. Here I got a higher accuracy with the LLM, but I think BERT is much better for such a task. Mainly for two reasons, the resources required and the fact that BERT always follows what we want. With BERT we will always get a nice output from the [CLS] token, instead with an LLM it's not certain that it will follow what we ask for in the prompt (for example, starting the answer with the correct choice).
Here we can look at a specific example, the sample 35 from the test dataset:

> fact1: All cnidarians are aquatic.
> fact2: Cnidarians include jellyfish and anemones.
> Question: What kind of animal are jellyfish?
> (A) protozoa (B) adult (C) paramecium (D) dry
> (E) land-based (F) Porifera **(G) anemones** **(H) aquatic**
> Correct answer: H

I find interesting this example, because given the facts is a very simple question.

> fact1: cnidarians → aquatic
> fact2: jellyfish → cnidarians
> ==> jellyfish → aquatic

Both models responded with G (anemones). It seems that the models were tricked by the presence of 'anemones' after the word jellyfish. It may be that 'anemones' is such a rare word, and that it appears in some kind of relationship with the word 'jellyfish', that it attracts the model's attention. It just sounds good to say 'jellyfish are anemones'. If I take this example and change fact2 by swapping the words 'anemones' and 'jellyfish', as follows:

> fact2 = 'Cnidarians include anemones and jellyfish'

I get the correct answer with the LLM (H), but I still get the wrong one with BERT. Perhaps because the LLM is a decoder-only architecture, with masked self-attention, and putting the word 'anemones' before 'jellyfish' has less of an effect on the prediction.

Conclusion

To complete the project, I began by implementing simple, foundational methods in NLP, knowing they would not produce state-of-the-art results but would still be valuable for understanding the evolution of the field. I started with TF-IDF, a basic yet effective method for capturing the importance of words in a document, and experimented with bigram/trigram language models.

Next, I explored neural models for generating word embeddings, such as Word2Vec, which is trained to predict the context of a given word. For sentence similarity, I primarily used cosine similarity to compare vector representations from both TF-IDF and word embeddings, selecting the alternative with the highest cosine similarity score.

Finally, I implemented BERT and large language models (LLMs) that utilize the transformer architecture, which is currently state-of-the-art for sequence-to-sequence tasks. Specifically, I examined the encoder (BERT-like models are encoder-only) that extracts meaning from input sequences and the decoder (LLMs are decoder-only) that generates new sequences based on the input.

I wanted to thank you for the lessons provided and Dr. Marco Braga for answering my questions about the project.

AI policies

I have mainly used **Copilot** with the extension in VSCode and sometimes **ChatGPT**. All the prompts I used can be found in the file ***"AI_prompts.txt"***.

I found Copilot very useful for speeding up simple things, for example it comes very handy for libraries like matplotlib, numpy and pandas, where maybe I just want to do a simple operation, but I can't remember the specific command/function. It is also quite handy for auto-completing the lines. I've used ChatGPT instead for more complex questions, usually when the code doesn't work. Beyond the coding aspect, I find it very useful to revise the text, not just to make it more formal but also to clarify the content.

In general I think that AI assistance can help speed up the process for simple things, without making a lot of mistakes. Instead, for more complex tasks it sometimes works and sometimes does not, but I think it is still useful as a last resort, trying it can be of help.