Natural Language Processing **Tel Aviv University**

Assignment 1: Word Vectors

Due Date: *June 27 , 2024* Lecturer: Maor Ivgi, TA: Noa Mark

Preliminaries

Submission Instructions This assignment is composed of both theoretical questions and code implementation ones. Your answers to the theoretical questions should be submitted in a single PDF, and should be written in digital format (we recommend using **overleaf** as you did in HW0 but you can use any digital editor you prefer). Code answers will be divided into two submission formats - in Section 5 and Section 6 you'll submit an **.ipynb** file (as you did in HW0), whereas for Section 2 you'll submit python files as part of your zip (you will fill in the missing code according to the instructions there, within the files that are attached to this assignment). The rest of the sections include theoretical questions only.

Submit your solution through Moodle. Your submission should consist of a single zip file named `<id1>_<id2>.zip` (where `id1` refers to the ID of the first student). This zip file should include the code and data necessary for running the tests provided out-of-the-box, as well as a written solution, the `.ipynb` notebook from sections 1 and 3 and 6, and the generated files `saved_params_40000.npy` and `word_vectors.png`.

Notes:

- Only one student needs to submit.
- Your code will be tested on the School of Computer Science operating system, installed on nova and other similar machines. Please make sure your code runs there. If your code does not run on nova, you code will not be graded.
- The provided tests are not exhaustive. Later parts of the assignment will reference this code so it is important to have a correct implementation.
- Your implementation should also be efficient and vectorized whenever possible (i.e., use numpy matrix operations rather than `for` loops). A non-vectorized implementation will not receive full credit!
- Your could should follow coding standards (well-documented, self-explained, meaningful naming, PEP8).

Acknowledgements This assignment was adapted from Stanford's CS224n course. Their contributions are greatly appreciated. In this note, we also like to acknowledge the work of the previous TAs of this course for building this exercise.

1 Understanding word2vec (theoretical)

Let's have a quick refresher on the `word2vec` algorithm.

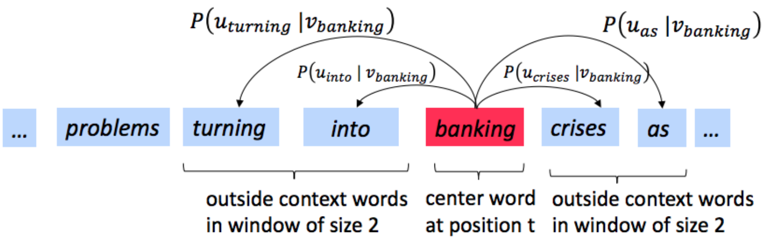

Figure 1: The word2vec skip-gram prediction model with window size 2

There is no need to submit this part. It's meant for you to verify your understanding (if you are not sure why learning it is significant here is some *motivation*). You will need close form of the partial derivative to implement the following section (you are not allowed to use automatic differentiation packages here). For further reading: *Stanford cs224n gradient notes*.

The key insight behind `word2vec` is that '*a word is known by the company it keeps*'. Concretely, suppose we have a 'center' word c and a contextual window surrounding c. We shall refer to words that lie in this contextual window as 'outside words'. For example, in Figure 1 we see that the center word c is 'banking'. Since the context window size is 2, the outside words are 'turning', 'into', 'crises', and 'as'.

The goal of the skip-gram `word2vec` algorithm is to accurately learn the probability distribution $P(O \mid C)$. Given a specific word o and a specific word c, we want to calculate $P(O = o \mid C = c)$, which is the probability that word o is an 'outside' word for c, i.e., the probability that o falls within the contextual window of c.

In word2vec, the conditional probability distribution is given by taking vector dot-products and applying the softmax function:

$$P(O = o \mid C = c) = \frac{\exp\left(\boldsymbol{u}_o^\intercal \boldsymbol{v}_c\right)}{\sum_{w \in \mathrm{W}} \exp\left(\boldsymbol{u}_w^\intercal \boldsymbol{v}_c\right)} \tag{1}$$

Here, W is the vocabulary, $\boldsymbol{u}_o$ is the 'outside' vector representing outside word o, and $\boldsymbol{v}_c$ is the 'center' vector representing center word c. To contain these parameters, we have two matrices, $\boldsymbol{U}$ and $\boldsymbol{V}$. The rows of $\boldsymbol{U}$ are all the 'outside' vectors $\boldsymbol{u}_w$. The rows of $\boldsymbol{V}$ are all of the 'center' vectors $\boldsymbol{v}_w$. Both $\boldsymbol{U}$ and $\boldsymbol{V}$ contain a vector for every $w \in W$[1].

Recall from lectures that, for a single pair of words c and o, the loss is given by:

$$\boldsymbol{J}_{\text{naïve-softmax}}(c, o, \boldsymbol{V}, \boldsymbol{U}) = -\log P(O = o \mid C = c) \tag{2}$$

Another way to view this loss is as the cross-entropy[2] between the true distribution $\boldsymbol{y}$ and the predicted distribution $\hat{\boldsymbol{y}}$. Here, both $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are vectors with length equal to the number of words in the vocabulary ($|W|$). Furthermore, the k^{th} entry in these vectors indicates the conditional probability of the k^{th} word being an 'outside word' for the given c. The true empirical distribution $\boldsymbol{y}$ is a one-hot vector with a 1 for the true outside word o, and 0 everywhere else. The predicted distribution $\hat{\boldsymbol{y}}$ is the probability distribution $P(O \mid C = c)$ given by our model in Equation (1).

In this section we're going to view many different derivatives. These derivatives will be used in the next

[1]Assume that every word in our vocabulary is matched to an integer number k. $\boldsymbol{u}_k$ is both the k^{th} row of $\boldsymbol{U}$ and the 'outside' word vector for the word indexed by k. $\boldsymbol{v}_k$ is both the k^{th} row of $\boldsymbol{V}$ and the 'center' word vector for the word indexed by k. **In order to simplify notation we shall interchangeably use k to refer to the word and the index-of-the-word**.

[2]The Cross Entropy Loss between the true (discrete) probability distribution p and another distribution q is $-\sum_i p_i \log(q_i)$.

section, where you'll be tasked to implement this algorithm (so you will need to have a closed-form of the derivative, as we're not using automatic differentiation packages here).

(a) Equation (1) uses the softmax function. Recall you should in HW0 that softmax is invariant to constant offset in the input, i.e that for any input vector $\boldsymbol{x}$ and any constant c,

$$\text{softmax}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} + c)$$

where $\boldsymbol{x} + c$ means adding the constant c to every dimension of $\boldsymbol{x}$. Remember that

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

(b) Verify you understand why the naïve softmax loss given in Equation (2) is the same as the cross-entropy loss between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$, i.e., show that

$$-\sum_{w \in \text{W}} y_w \log(\hat{y}_w) = -\log(\hat{y}_o) \tag{3}$$

(c) Recall that in class we showed that the partial derivative of $\boldsymbol{J}_{\text{naïve-softmax}}(c, o, \boldsymbol{V}, \boldsymbol{U})$ with respect to $\boldsymbol{v}_c$ is $\boldsymbol{u}_o - \mathbb{E}_{o' \sim p(o'|c)}[\boldsymbol{u}_{o'}]$. Now, Compute the partial derivative of $\boldsymbol{J}_{\text{naïve-softmax}}(c, o, \boldsymbol{V}, \boldsymbol{U})$ with respect to each of the 'outside' word vectors, $\boldsymbol{u}_w$'s. There will be two cases: when $w = o$, the true 'outside' word vector, and when $w \neq o$, for all other words.

You should verify you understand why the answer is as follow:

$$\frac{\partial}{\partial u_{w \neq o}} \mathbf{J}_{naive-softmax}(c, o, \mathbf{V}, \mathbf{U}) = \hat{y_w} v_c$$

$$\frac{\partial}{\partial u_o} \mathbf{J}_{naive-softmax}(c, o, \mathbf{V}, \mathbf{U}) = v_c(y^T \hat{y} - 1)$$

Now we shall consider the negative sampling loss, which is an alternative to the naïve softmax loss. Assume that K negative samples (words) are drawn from the vocabulary W. For simplicity of notation we shall refer to them as $w_1, w_2, \ldots, w_K$ and to their corresponding outside vectors as $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_K$. Note that $o \notin \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_K\}$. For a center word c and an outside word o, the negative sampling loss function is given by:

$$\boldsymbol{J}_{\text{neg-sample}}(c, o, \boldsymbol{V}, \boldsymbol{U}) = -\log(\sigma(\boldsymbol{u}_o^\intercal \boldsymbol{v}_c)) - \sum_{k=1}^{K} \log(\sigma(-\boldsymbol{u}_k^\intercal \boldsymbol{v}_c))$$

for a sample $w_1, w_2, \ldots, w_K$, where $\sigma(\cdot)$ is the sigmoid function[3].

The partial derivative of this loss with respect to $\boldsymbol{v}_c$ is: $-(1 - \sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c))u_o + \sum_{k=1}^{K} (1 - \sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c))\boldsymbol{u}_k$.

Now, with respect to $\boldsymbol{u}_o$: $-(1 - \sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c))v_c$.

And with respect to $\boldsymbol{u}_k$: $(1 - \sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c))v_c$.

You'll use these derivatives while implementing the algorithm in next section.

[3]The loss function here is the negative of what Mikolov et al. had in their original paper, because we are minimizing rather than maximizing in our assignment code. Ultimately, this is the same objective function.

(d) Suppose the center word is $c = w_t$ and the context window is $[w_{t-m}, \ldots, w_{t-1}, w_t, w_{t+1}, \ldots, w_{t+m}]$, where m is the context window size. Recall that for the skip-gram version of `word2vec`, the total loss for the context window is:

$$\boldsymbol{J}_{\text{skip-gram}}(c, w_{t-m}, \ldots, w_{t+m}, \boldsymbol{V}, \boldsymbol{U}) = \sum_{\substack{-m \le j \le m \\ j \ne 0}} \boldsymbol{J}(c, w_{t+j}, \boldsymbol{V}, \boldsymbol{U})$$

Here, $\boldsymbol{J}(c, w_{t+j}, \boldsymbol{V}, \boldsymbol{U})$ represents an arbitrary loss term for the center word $c = w_t$ and outside word w_{t+j}. $\boldsymbol{J}(c, w_{t+j}, \boldsymbol{V}, \boldsymbol{U})$ could be $\boldsymbol{J}_{\text{naïve-softmax}}(c, w_{t+j}, \boldsymbol{V}, \boldsymbol{U})$ or $\boldsymbol{J}_{\text{neg-sample}}(c, w_{t+j}, \boldsymbol{V}, \boldsymbol{U})$, depending on your implementation.

Write down three partial derivatives:

(i) $\partial \boldsymbol{J}_{\text{skip-gram}}(c, w_{t-m}, \ldots, w_{t+m}, \boldsymbol{V}, \boldsymbol{U}) / \partial \boldsymbol{U}$

(ii) $\partial \boldsymbol{J}_{\text{skip-gram}}(c, w_{t-m}, \ldots, w_{t+m}, \boldsymbol{V}, \boldsymbol{U}) / \partial \boldsymbol{v}_c$

(iii) $\partial \boldsymbol{J}_{\text{skip-gram}}(c, w_{t-m}, \ldots, w_{t+m}, \boldsymbol{V}, \boldsymbol{U}) / \partial \boldsymbol{v}_w$ when $w \ne c$

Write your answers in terms of $\partial \boldsymbol{J}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U}) / \partial \boldsymbol{U}$ and $\partial \boldsymbol{J}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U}) / \partial \boldsymbol{v}_c$.

You should verify you understand why the answer is as follow:

(i)

$$\sum_{-m \le j \le m} \frac{\partial \mathbf{J}(v_c, w_{t+j}, V, U)}{\partial U}$$

(ii)

$$\sum_{-m \le j \le m} \frac{\partial \mathbf{J}(v_c, w_{t+j}, V, U)}{\partial v_c}$$

(iii)

$$0$$

Once you're done: *Given that you computed the derivatives of $\boldsymbol{J}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U})$ with respect to all the model parameters $\boldsymbol{U}$ and $\boldsymbol{V}$ in parts (a) to (c), you have now computed the derivatives of the full loss function $\boldsymbol{J}_{\textit{skip-gram}}$ with respect to all parameters. You're ready to implement* `word2vec`*!*

(e) Try to explain in a few sentences why it is important to split each token representation into two - first for its being the center token, and second for it being an output token.

(f) Finally, try to explain the intuition behind this algorithm. That is, why we might expect this algorithm to lead the model end up representing two semantically similar tokens with two "close" vectors within the Euclidean space.

2 Implementing word2vec (code implementation)

In this part you will implement the word2vec model (using the knowledge you gained in the previous section) and train your own word vectors with stochastic gradient descent (SGD).

Before you begin, first run the following commands within the assignment directory in order to create the appropriate conda virtual environment. This guarantees that you have all the necessary packages to complete the assignment.

```
conda env create --file env.yml
conda activate nlp-hw1
```

Once you are done with the assignment you can deactivate this environment by running:

```
conda deactivate
```

Note: you are not required to use conda in this assignment, but the usage of virtual environment managers (such as conda) is highly recommended and is standard practice in both the industry and the academia. If you are unfamiliar with conda, check out `https://docs.conda.io/projects/conda/en/latest/user-guide/getting-started.html`.

(a) Implement the `softmax` function in the module `q2a_softmax.py`. Note that in practice, for numerical stability, we make use of the property we proved in question 1.a and choose $c = -\max_i x_i$ when computing softmax probabilities (i.e., subtracting its maximum element from all elements of x). You can test your implementation by running `python q2a_softmax.py`.

(b) To make debugging easier, we will now implement a gradient checker. Fill in the implementation for the `gradcheck_naïve` function in the module `q2b_gradcheck`. You can test your implementation by running `python q2b_gradcheck.py`.

(c) Fill in the implementation for `naïve_softmax_loss_and_gradient`, `neg_sampling_loss_and_gradient`, and `skipgram` in the module `q2c_word2vec.py`. You can test your implementation by running `python q2c_word2vec.py`. Verify that your results are approximately equal to the expected results.

(d) Complete the implementation for the SGD optimizer in the module `q2d_sgd.py`. You can test your implementation by running `python q2d_sgd.py`.

(e) Show time! Now we are going to load some real data and train word vectors with everything you just implemented! We are going to use the Stanford Sentiment Treebank (SST) dataset to train word vectors, and later apply them to a simple sentiment analysis task. There is no additional code to write for this part; just run `python q2e_run.py`.

Note: The training process may take a long time depending on the efficiency of your implementation. Plan accordingly!

After 40,000 iterations, the script will finish and a visualization for your word vectors will appear. It will also be saved as `word_vectors.png` in your project directory. **Include the plot in your homework write up, inside the pdf (not a separate file)**. Briefly explain what you notice in the plot. Are there any reasonable clusters/trends? Are the word vectors as good as you expected? If not, what do you think could make them better?

3 Optimizing word2vec (theoretical)

We will now prove that in the skipgram algorithm, the maximum likelihood solution for word embedding probabilities is their empirical distribution, and show that there exists a scenario where reaching the optimum is impossible.

(a) For a corpus of length T, recall that the objective is:

$$\mathcal{L}(\theta) = \prod_{t=1}^{T} \prod_{-m\leq j\leq m, j\neq 0} p_\theta\left(w_{t+j} \mid w_t\right)$$
$$J(\theta) = \log \mathcal{L}(\theta) = \sum_{t=1}^{T} \sum_{-m\leq j\leq m} \log p_\theta\left(w_{t+j} \mid w_t\right)$$

Prove that if $\theta^* = \arg\max_\theta \mathcal{L}(\theta)$ then $p_{\theta^*}(o \mid c) = \frac{\#(c,o)}{\sum_{o'} \#(c,o')}$.
Where $\#(c, o) =$ Number of co-occurrence of c and o in the corpus.

Hint 1: For a fixed c, o in the vocabulary, how many times does the term $p_\theta(o \mid c)$ appear in $\mathcal{L}(\theta)$
Hint 2: Use Lagrange multipliers.

(b) Let's assume each word is represented by a single scalar (real number). Prove that there is a corpus over a vocabulary of no more than 4 words, where reaching the optimum solution is impossible. You can assume that a corpus is a list of sentences, such as $\{$"*aa*", "*bb*", "*cc*", $\ldots, \}$. For the given corpus: $\{aa, aa, aa, ab, ab, ac\}$. We have:

$$p(a|a) = 0.5$$
$$p(b|a) = 1/3$$
$$p(c|a) = 1/6$$

4 Paraphrase Detection (theoretical)

Paraphrase detection is a binary classification task, where given two sentences, the model needs to determine if they are paraphrases of one another. For example the pair of sentences *"The school said that the buses accommodate 24 students"* and *"It was said by the school that the buses seat two dozen students"* are paraphrases. While the pair *"John ate bread"* and *"John drank juice"* are not.

Let's denote by x_1, x_2 the input pair of sentences and by $\mathbf{x}_1, \mathbf{x}_2$, a vector representation for the pair of sentences obtained using some neural network, where $\mathbf{x}_i \in \mathbb{R}^d$.

Consider the following model for paraphrase detection:

$$p(\text{the pair is a paraphrase} \mid x_1, x_2) = \sigma\left(\text{relu}(\mathbf{x}_1)^\top \text{relu}(\mathbf{x}_2)\right),$$

where $\text{relu}(x) = \max(0, x)$.

(a) In this model, what is the maximal accuracy on a dataset where the ratio of positive to negative examples is 1:4?

(b) Suggest a simple fix for the problem.

(c) What evaluation metrics should you use to evaluate the success of a model on this imbalanced dataset? Please explain why. Consider: Accuracy, precision, Recall, ROC-AUC, AUC-PR, confusion matrix.

5 TF-IDF (code implementation)

TF-IDF stands for Term Frequency Inverse Document Frequency of records. It is a common measure, usually used to assess how relevant a word (which usually appears within a given sequence) is to a given text. This metric is extensively used in tasks such as Information Retrieval (IR) - given a query (might be an NL question, an SQL sequence, a structured pattern of keywords, etc.), the goal is to search the most relevant context passages out of a large corpus consisting of many of them, in terms of the corresponding downstream task. An important example comes from Question Answering - many modern QA systems are a compound of an information retrieval model, and a reading comprehension (RC) model. The purpose of the IR model is: given the question that has been asked, to find the most relevant passages appearing in the web (e.g. Wikipedia), which are likely to contain the answer. Then, the purpose of the RC model is to extract the specific answer out of them.

Later in this course, you will be learning about IR more extensively. This exercise's goal is to give a first exposure to IR and to get you be familiar with one of its most fundamentals algorithms - the $TF - IDF$. Through this exercise you will fully implement this algorithm, and also see a real use-case of it.

Please upload the *tfidf.ipynb* file into colab, and implement the code according to the next instructions.

We will use t to refer a single token/word. C will stand as our corpus, which is formally a set of documents. We will use d to denote a single document from the corpus.

1. Term Frequency(TF). Suppose we have a set of English text documents and wish to rank which document is most relevant to the query , "What is the capital of New Zealand?". A simple way to start out is by eliminating documents that do not contain all three words "capital","New", and "Zealand", but this might still leave many irrelevant documents (depending on our corpus). To further distinguish between them, we might count the number of times each term occurs in each document; the number of times a term occurs in a document is called its term frequency, and denoted by TF. Formally, if we denote the number of times t appears in d as $count(t, d)$, and the total number of tokens in d by $size(d)$, we get: $TF(t, d) = \frac{count(t,d)}{size(d)}$.

   Implement the function `computeTF`, which given a token and a document, computes the corresponding TF measure.

2. Document Frequency (DF). This measures the importance of document in whole set of corpus, this is very similar to TF. The only difference is that TF is frequency counter for a term t in document d, where as DF is the count of occurrences of term t in the document set C. In other words, DF is the number of documents in which the word is present. Formally, $DF(t, N) = \sum_{d \in N} 1_{t \in d}$.[4] We add one occurrence if the term appears in a document at least once, we do not need to know the number of times the term appears.

   Implement the function `computeDF` which given a token, and a corpus (which is a list of documents), computes the token's DF.

3. Inverse Document Frequency (IDF). While computing TF, all terms are considered equally important. However it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scaling up the rare ones which may be more indicative of related documents when they co-occur, by computing the IDF, an inverse document frequency factor is incorporated which diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. IDF is the inverse of the document frequency which measures the informativeness of term t. When we calculate IDF, it will be very low for the most frequent words such as stop words (because stop words such as "is" is present in almost all of the documents and thus give very little information when they appear, and $\frac{size(C)}{DF}$ will assign a very low value for that word). This finally gives what we want, a relative weighting. Formally, we would like to define it as: $IDF(t) = \frac{size(C)}{DF(t)}$ Now there are few other problems with this, in case of a large corpus, say one with 100,000,000 documents, the IDF value explodes. To avoid the effect we take the log of IDF. During the query time, when a word which is not in vocabulary appears, the DF will be 0. As we cannot divide

[4] 1_b is 1 when the boolean condition b is true, and 0 otherwise.

by 0, we smooth the value by adding 1 to the denominator. Thus, we arrive to the final formula: $IDF(t) = \log \frac{size(C)}{DF(t)+1}$.

Implement the function `computeIDF` which given a token and a corpus, computes the corresponding IDF of the token.

4. We are finally ready to compute $TF - IDF$, our measure that evaluates how important a word is to a document in a corpus. There are many different variations of $TF - IDF$ but for now let us concentrate on the this basic version. $TFIDF(t, d) = TF(t, d) * IDF(t, C)$.

   Implement the function `computeTFIDF` which given a token, a document and a corpus, computes the corresponding $TFIDF$ of the token and the document.

5. Now you're going apply $TF - IDF$ on a case that might have been a real use-case for Question Answering systems. First, run the cells of code which construct the corpus, and define the query. Then, add code of your own that uses the $TF - IDS$ algorithm you already implemented (you may edit/add new functions of course), in order to retrieve the most relevant sequences out of the corpus, to the query: *What is the capital of New Zealand?*. As a final answer, please print the 3 highest scored sentences out of the corpus.

   This is a fun tip, you will not be graded on that: You may also keep messing around with this code. Specifically, you may try to split the passage differently (namely, not necessarily sentence-based separation), test different queries, and consider additional passages from Wikipedia to enriching the corpus.

6 Topic modeling (practical)

In this exercise, you will be asked to browse solutions for a specific downstream task in NLP. The goal of this exercise, is to get you comfortable with using others' research code and common tools. We hope that this practical experience will help you towards your final project. We encourage you to continue diving deeper, employing critical thinking and creative ideas!

In the task of **Topic Modeling** [5] we try to find some underlying semantic structure in a dataset using statistical methods. For example, we can cluster documents into topics with unsupervised methods. This tool can be used for data-analysis and data mining, with applications in NLP, Bio-informatics, Software Engineering, Social networks and so on. Some popular and traditional algorithms for this task are based on TF-IDF (learned in Section 5).

Complete the code in the *colab notebook* (not a lot of lines to complete, don't worry) and answer the following question in your pdf (do not forget to submit your version of this notebook):

1. *Paper with code* is free and open resource platform with Machine Learning papers, code, datasets, methods and evaluation tables (by Meta AI Research). You can browse state-of-the-art solution to different tasks according to categories; get familiar with thousands of benchmarks datasets; compare models with varied evaluation metrics and so on.

[5] When getting into a new research subject, "review papers" are a good place to start. They provide an introduction and a literature overview (See *review1*, *review2* for example. These are intended for personal enrichment only and are not obligatory. You can access using the university account). Note that reviews are not always up-to-date, especially in a field that advances as fast as NLP. Hence, searching the state-of-the-art using Google Scholar or other search tools is usually required.

Explore *topic modeling on text challenge* using the task leaderboard and answer the following questions:

(a) What are the metrics used to evaluate Topic modeling (name at least two, not including perplexity)? Please explain each of them (Feel free to use the web to extend your knowledge).

(b) What benchmark datasets exist for this task? (name at least 3).

2. *HuggingFace* is a platform for machine learning and AI, providing open source python libraries, publicly available model weights, datasets and forums for the AI community. You will learn more about it in a dedicated tutorial in the future. In this exercises we will use and explore very specific and limited set of features from this platform: loading a dataset and a trained model.

   (a) Complete and run the code in "Loading dataset" section in the notebook by providing a reference for 20_newsgroups dataset. You can find the dataset *here*.

   (b) BERTopic is a topic modeling algorithm that uses TF-IDF with transformers (don't worry you will learn more about transformers in the future, stay tuned! You can read *BERTopic paper* and

   Complete and run the code in "Loading BERTopic model" section in the notebook by providing a model from Hugging Face hub. Search for a BERTopic [6] model trained on 20_newsgroups dataset *here*.

   (c) Now, we will evaluate the model performance on the benchmark dataset. We will use the gensim package for topic modeling evaluation. Complete and run the code in "Evaluation using Gensim" section. Note, you need to fill in the coherence metrics names (same as Q1.a). Verify the string format match the expected input to CoherenceModel according to the library documention.

   Write in the pdf the results: the model, the evaluation metrics names and their values.

   (d) Can you think of some aspect of the task that these metrics fail to evaluate, but human-evaluator may consider? Write your thoughts in the pdf.

   (e) (Bonus:) If you are responsible for evaluating a new topic-modeling solution, how would you approach this task?

[6] More about BERTopic library integration with Hugging Face in *here*