Department of Information Engineering
Artificial Intelligence and Data Engineering
Business and Project Management

Terms of Services Summarization

E. Ruffoli, F. Hudema, T. Baldi

Academic Year 2021/2022

Contents

1 Introduction: Terms Of Service transparency problem

Today one of the hottest topics in the digital industry is **privacy**. As pointed out in an article in The Guardian newspaper, the problem of *click-to-agree* contracts is a serious social phenomenon that should not be neglected. Users give the right to keep, analyze and sell their data to web-based services and third parties by accepting contracts about which they are not aware of. **Increasingly often, people click away their right to go to court if anything goes wrong**.

As described in the previous article, two communication professors, Jonathan Obar of York University in Toronto and Anne Oeldorf-Hirsch of the University of Connecticut, had carried out a social experiment on college students in which they click the great green "Join" button to become members of *NameDrop*, a new social network. According to paragraph 2.3.1 of the terms of service, they had agreed to give *NameDrop* their future firstborn. Fortunately, *NameDrop* does not exist, but this experiment underline the fact that nobody reads on-line contracts, license agreements, terms of service, privacy policies and other agreements.

The *no-reading* problem is not new. After all few people read the fine print even when it was literally in print, but it is possible that the design of click-to-accept pages makes the problem worse. A possibile solution to address this problem may be providing an automated synthesis system that will extract key information from these contracts in order to make them more understandable and show them in a more convenient way to the user.

Artificial Intelligence may be the key to develop such synthesis system. The branch of AI, that is focused on text and language processing is called NLP and today is certainly one on which more research is carried out. Thanks to these modern technologies, software are able to break down and interpret human language: many tasks can be faced with NLP such as translation, chatbots, spam filters and **text summarization**.

2 Text Summarization

Summarization is the task of condensing a piece of text to a shorter version, reducing the size of the initial text while at the same time preserving key informations and the general meaning of the content. There are two different approaches for text summarization: **extractive** and **abstractive**.

2.1 Extractive Summarization

The extractive approach involves selecting the most important sentences from the documents **as they are** and then concatenating all of them to generate the summary. In other terms, every phrase and word of the summary actually belongs to the original document which is summarized.

2.2 Abstractive Summarization

Abstractive summarization methods are more complex since they aim at generating a summary by interpreting and understanding the text. They use advanced natural language techniques in order to generate a **new** shorter text: the summary text may contain parts that do not appear in the original document and that are generated by rephrasing sentences and incorporating information. An acceptable abstractive summary covers core information in the original text and is linguistically fluent.

2.3 Terms of Service Summarization

The goal of this project is to produce an abstractive summarization model capable of summarizing legal contracts, in particular terms of service ones. We have opted for an abstractive summarization model because we thought that in order to make the contract more comprehensible it is necessary to replace technical terms and expressions with more commonly used ones.

As said earlier, this type of summarization requires advanced machine learning techinques. In detail this is a classic sequence-to-sequence (seq2seq) task with an input text and a target text. This is the perfect scenario for *encoder-decoder transformers*, a particular machine learning architecture introduced in 2017 in the paper "Attention is all you need", that turned out to be a gamechanger for natural language processing.

3 Background

In this section will be showcased the preliminary concepts needed to fully understand the idea behind transformers.

3.1 Encoder-Decoder Framework

Prior to the introduction of the transformers, recurrent neural networks (RNNs) such as LSTMs were the state of the art for NLP tasks.

In these architectures to process the nth token (word), the model combines the vector representing the sentence up to token *n-1* with the information of the new token to create a new vector, representing the sentence up to token n.

Typically, the RNNs realized for NLP tasks usually presents an **encoder-decoder** architecture: the job of the encoder is to encode the information from the input sequence into a numerical representation that is often called the *last hidden state*; this state is then passed to the decoder, which generates the output sequence. The *transformers* encoder-decoder architecture will be analyzed in detail in the next section.

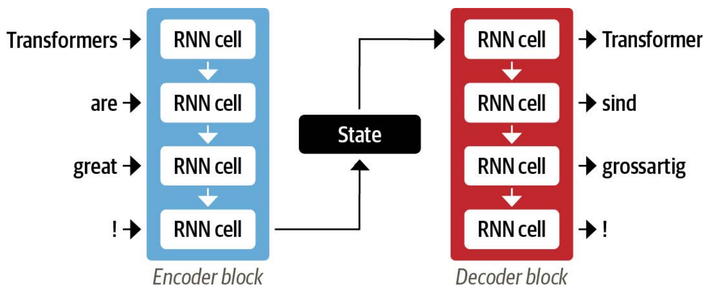


Figure 1: An encoder-decoder architecture with a pair of RNNs for a language translation task [2]

However, approaching NLP tasks with RNNs has brought out two major weaknesses:

- the final hidden state of the encoder creates an information bottleneck since it has to represent the meaning of the whole input sequence because this is all the decoder has access to when generating the output.
- the dependency of token computations on results of previous token computations makes it hard to parallelize computation on modern deep learning hardware leading to inefficient RNNs training;

3.2 Attention

In the encoder-decoder architecture of RNNs, the *last hidden state* of the encoder creates an **information bottleneck** since it has to represent the meaning of the whole input sequence.

To overcome this, there is a way to allow the decoder to have access to all of the encoder's hidden states: *attention*. The main idea is that instead of producing a single hidden state for the input sequence, **the encoder outputs a hidden state at each step that the decoder can access**. The attention mechanism lets the decoder assign a different amount of weight, or "attention," to each of the encoder states at **every decoding timestep** in order to prioritize which states to use.

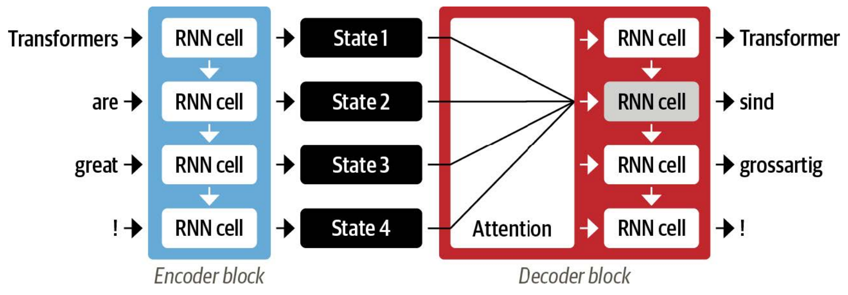


Figure 2: An encoder-decoder architecture with an attention mechanism for a pair of RNNs for a language translation task [2]

In figure 3 is shown how the attention mechanism works: it remarks at which part of the input sentence we're paying attention to at each decoding step.

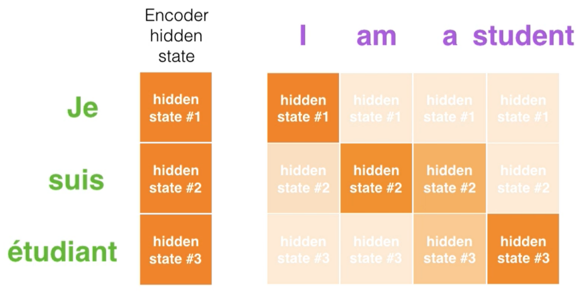


Figure 3: Attention example of a French to English translation task[3]

4 Transformers

As mentioned, the original Transformer, proposed in the paper, is based on the *encoder-decoder* architecture. Actually, it is quite common to have transformers that have only the encoder component or only the decoder component but, for summarization tasks, the complete architecture with both the components is the most used. In figure 4 it is shown the transformer architecture introduced in the paper.

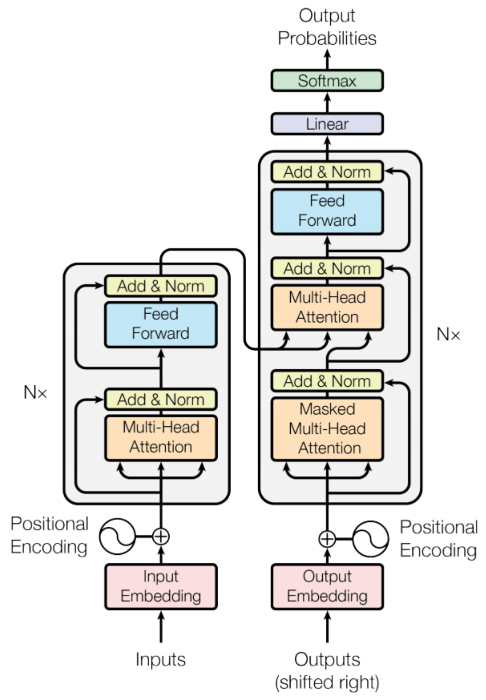


Figure 4: The transformer model architecture [1], the encoder is the left part while the decoder is the right one.

4.1 Encoder

Recall that the encoder job is to convert an input sequence of tokens into a sequence of embedding vectors, often called the hidden state or context. The encoding component is actually a stack of encoders (6 in the original paper), each of which has two sub-layers: the first sub-layer is a *multi-head self-attention layer*, while the second is a fully connected *feed-forward network*.

The encoder's inputs first flow through the self-attention layer, whose job is to help the encoder look at other words in the input sentence as it encodes a specific word; then the outputs of the self-attention layer are fed to the feed-forward neural network, that consists of two linear transformations with a ReLU activation in between.

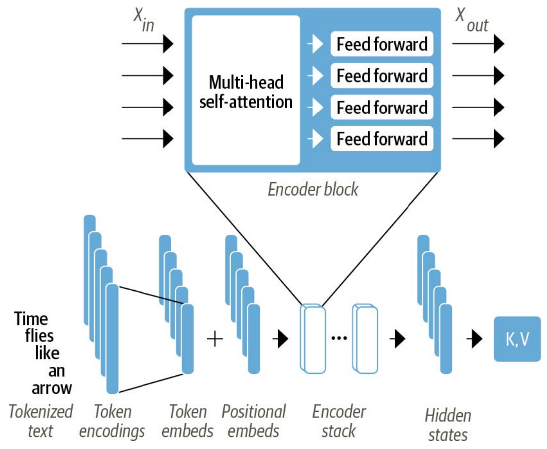


Figure 5: Encoder component for a translation task[2]

4.2 Self Attention

Transformers now do not require RNNs and instead use on a special form of attention called *self-attention*. As the model processes each word (each position in the input sequence), *self-attention* allows it to look at relevant positions in the input sequence that can help lead to a better encoding of the word.

In *self-attention*, the attention weights are computed for all the hidden states of the encoder; by contrast, the attention mechanism associated with RNNs involves computing the relevance of each encoder hidden state to the decoder hidden state at a given decoding timestep.

So to summarize the difference: traditional attention was used in combination with RNNs to improve their performance; ***self-attention* is used instead of RNNs**.

Formally, given a sequence of token embeddings $x_1, ..., x_n$, *self-attention* produces a sequence of new embeddings $x'_1, ..., x'_n$ where each x'_i is a linear combination of all the x_j:

$$x'_i = \sum_{j=1}^{n} w_{ji} x_j$$

The coefficients w_{ji} are the *attention weights* and are normalized so that the sum of the weights is equal to 1.

In figure 6 is shown how *self-attention* can generate two different representations for the word “flies” based on the context. It updates raw token embeddings (upper) into contextualized embeddings (lower) to create representations that incorporate information from the whole sequence.

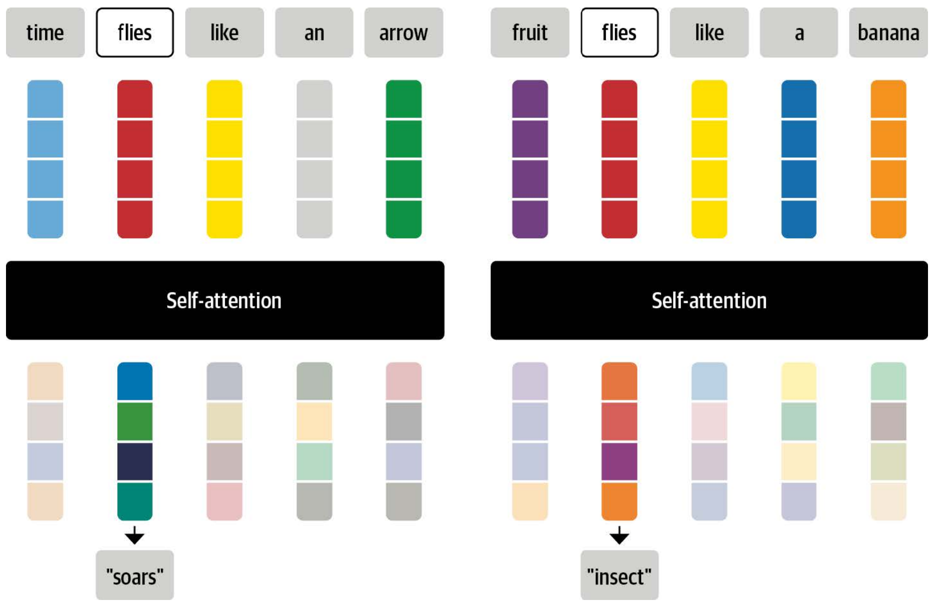


Figure 6: Self Attention example [2].

4.2.1 Scaled Dot-Product Attention

The *scaled dot-product attention* is a method, proposed in the ”Attention is all you need” paper, to compute *self-attention* in order to implement a self-attention layer. Here are the main steps:

1. For each word (token embedding), create three vectors, called **query**, **key**, and **value**, by multiplying the token embedding by three matrices that were trained during the training process.

2. For each word compute the **attention score** between every other word of the input sentence and the current word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position. The score is calculated by taking the dot product of the **query** vector with the **key** vector of the respective word we are scoring. So if we are processing the self-attention for the word in position 1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2 and so on. The output from this step, for a sequence with n input tokens, is a n $\times$ n matrix of attention scores.

3. Compute the **attention weights** by multiplying the attention scores by a scaling factor to normalize their variance and then normalized with a softmax to ensure all the column values sum to 1. The resulting n $\times$ n matrix now contains all the attention weights, w_{ji}.

4. Update the token embeddings. Once the attention weights are computed, we multiply them by the **valueù** vector $v_1, ..., v_n$ to obtain an updated representation of the embedding $x'_i = \sum_{j=1}^{n} w_{ij} x_j$

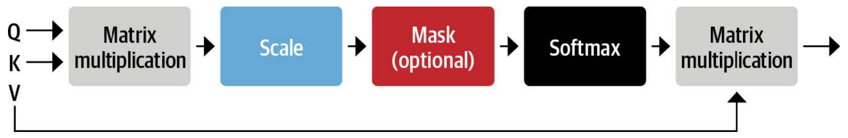


Figure 7: Scaled dot product attention workflow [2].

4.2.2 Multi Headed Attention

The "Attention is all you need" paper adds another mechanism to the self-attention layer, called **multi-headed attention**. It consists of several self-attention layers running in parallel and it significantly improves the comprehension capability of the attention layer.

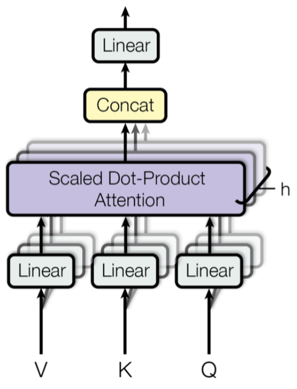


Figure 8: Multi-Head Attention [1].

4.3 Decoder

The decoder uses the encoder's hidden state to iteratively generate an output sequence of tokens, one token at a time. The decoding component of the transformer is also composed of a stack of decoders (6 in the original paper). In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, the *encoder-decoder attention layer*, which compute multi-head attention over the output (the attention vectors **key** and **value**) of the encoder stack.

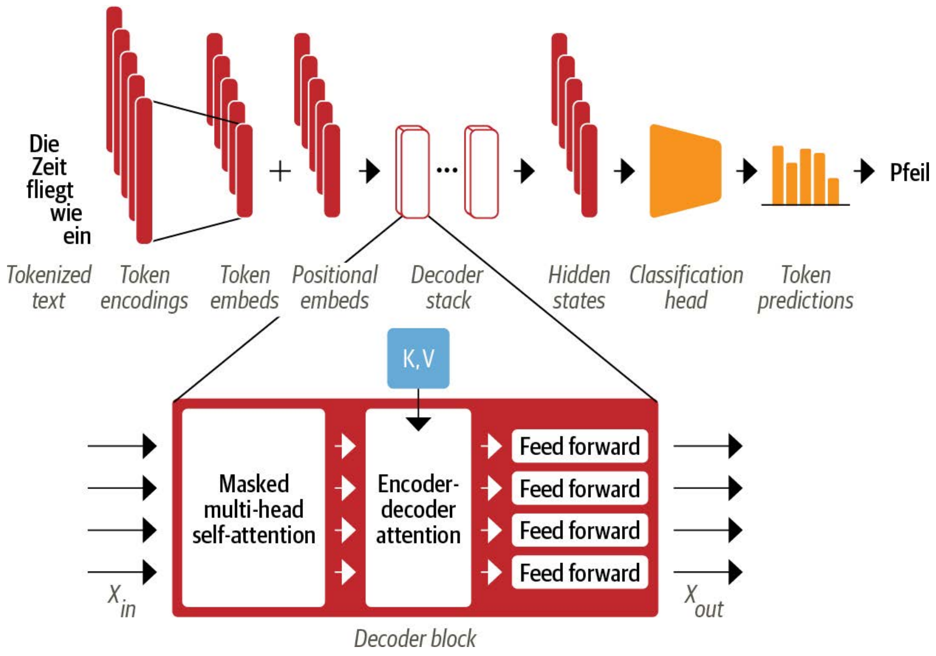


Figure 9: Decoder component for a translation task [2].

5 Dataset

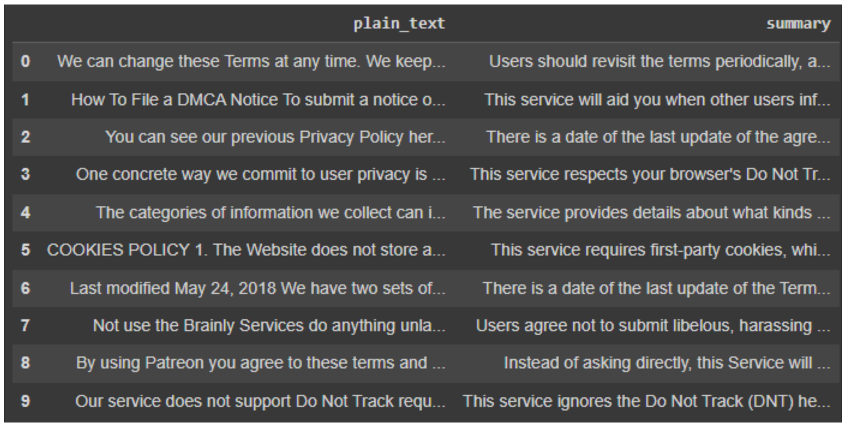
To build our terms of service summarizer we use a dataset scraped from *tosdr.org*, a project which aims to analyze and grade the terms of service and privacy policies of major Internet sites and services. This dataset allowed us to have a large number of documents that have been summarized with the supervision of domain experts. This was necessary to fine-tuning the model for the abstractive summarizer task.

5.1 Scraping and data preprocessing

We obtained the necessary data with a Python script exploiting the API endpoints provided by the site. The code is available in the *scraping* folder of the repository.

In the preprocessing phase we had to eliminate the documents that did not contain summary and clean the text of any special characters and html tags. Since not all documents were in the same language, we only selected those in English using *Spacy_language_detection*, a fully customizable language detection for spaCy pipeline.

We have created a dataset of 4MB containing 901 terms of service and privacy policies contracts relating to the main web services. Finally, we split the dataset into 3 parts, 80% as a training set and the remaining part in equal



	plain_text	summary
0	We can change these Terms at any time. We keep...	Users should revisit the terms periodically, a...
1	How To File a DMCA Notice To submit a notice o...	This service will aid you when other users inf...
2	You can see our previous Privacy Policy her...	There is a date of the last update of the agre...
3	One concrete way we commit to user privacy is ...	This service respects your browser's Do Not Tr...
4	The categories of information we collect can i...	The service provides details about what kinds ...
5	COOKIES POLICY 1. The Website does not store a...	This service requires first-party cookies, whi...
6	Last modified May 24, 2018 We have two sets of...	There is a date of the last update of the Term...
7	Not use the Brainly Services do anything unla...	Users agree not to submit libelous, harassing ...
8	By using Patreon you agree to these terms and ...	Instead of asking directly, this Service will ...
9	Our service does not support Do Not Track requ...	This service ignores the Do Not Track (DNT) he...

Figure 10: Dataset

proportion as a test and validation set.

Since those deep learning models currently have a limit on the number of tokens they can use ad input, before start the training we need to take a quick look at the length distribution of the texts and summaries.

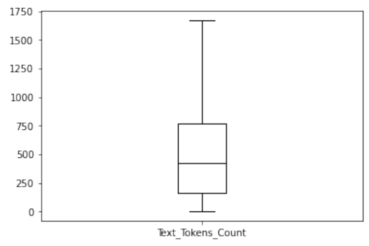


Figure 11: Text Tokens Count

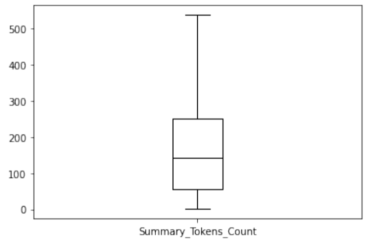


Figure 12: Summary Tokens Count

We see that most of the plain texts and summaries are shorter than 1024, the maximum input size these models have. We will consider as the maximum lengths 1024 for the next stages.

6 Hugging Face and model deployment

The *Hugging Face* ecosystem consist of two main parts: a family of **library** and the **Hub**. The library supports the major depp learning frameworks including PyTorch, and provides a standardized interface to a wide range of transofmer models.

The Hub allow to use models directly in code by downloading them from the hub and since transofmer models cannot receive raw string as input, but need that the text has beed tokenized and encoded as numerical vectors, offers also the implementation of different tokenizer associated with pretrained model. The tokenier can be load in the same way we can load pretrained model.

Example of using a pretrained model from Hugging Face Hub

```
from transformers import pipeline, AutoTokenizer

model = 'facebook/bart-large-cnn'
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)

pipe = pipeline("summarization", model = model, tokenizer=tokenizer,
    device=0)
pipe_out = pipe(sample_text)
```

In the next section, will be explained the steps on which model training and fine tuning are based. The complete code for training and model deployment can be found in the repository.

6.1 Fine Tuning with TOSDR Dataset

The main idea here is to use *pretrained model* as a initial point, in order to speed up training and improve performance, adapting the model *domain* to the use case. The general framework for fine-tuning of neural pretrained models for various tasks consists of the following steps:

1. Start from Transformer models (BART, T5, etc.) that have been trained as **language models**. This means they have been trained on large amounts of raw text in a *self-supervised* fashion. Self-supervised learning is a type of training in which the objective is automatically computed from the inputs of the model. That means that humans are not needed to label the data. This type of model develops a statistical understanding of the language it has been trained on, with the objective to predict the next word based on the previous words. The general strategy to achieve better performance is by increasing the models' sizes as well as the amount of data they are pretrained on.

2. *Pretraining* is the act of training a model from scratch. This pretraining is usually done on very large amounts of data. In our case, for performing text summarization, we selected pretrained models on English language

that has been adapted to the summarization task using datasets as the cnn_dailymail, containing just over 300k unique news articles as written by journalists at CNN and the Daily Mail, and the highlights of this articles.

3. *Fine-tuning*, on the other hand, is the training done after a model has been pretrained. To perform fine-tuning, it's necessary to start from a pretrained language model, then perform additional training with a dataset specific to the task. Since the pretrained model was already trained on lots of data, the fine-tuning requires way less data to get decent results. For the same reason, the amount of time and resources needed to get good results are much lower. We used the dataset scraped with term of services to perform a fine tuning of the pretrained model.

6.2 Google Colab

Since we're dealing with neural models and the training require high computational resources, we need access to a computer with an high-end NVIDIA GPU to perfom training. Google Colab is a hosted Jupyter notebook service that requires no setup to use, while providing access free of charge to computing resources including GPUs. The free plane provide the GPU accelaration with a NVIDIA T4 with 16GB of memory, that we use to trained the models used in this project.

7 Evaluation metrics

Good evaluation metrics are important, since we use them to measure the performance of models not only when we train them but also later, in production. If we have bad metrics we might be blind to model degradation, and if they are misaligned with the business goals we might not create any value.

Measuring performance on a text generation task is not as easy as with standard classification tasks such as sentiment analysis or named entity recognition. Simply checking for an exact match to a reference text is not optimal, even humans would fare badly on such a metric because we all write text slightly differently from each other.

Two of the most common metrics used to evaluate generated text are BLEU and ROUGE. In this project we use the ROUGE metric instead of the BLEU because it is preferible for the summarization task.

7.1 ROUGE

The ROUGE score was specifically developed for applications like summarization where high recall is more important than just precision. ROUGE is actually a set of metrics:

- *ROUGE-N*, it measures the number of matching n-grams[1] between our model-generated text and the reference. The N represents the n-gram that we are using. For ROUGE-1 we would be measuring the match-rate of unigrams between our model output and reference. ROUGE-2 and ROUGE-3 would use bigrams and trigrams respectively. Now we have to compute the **recall**, which is the number of overlapping n-grams found in both the model output and reference, then divides this number by the total number of n-grams in the reference:

  $$\frac{\textit{number of n-grams found in model and reference}}{\textit{number of n-grams in reference}}$$

  This is great for ensuring our model is capturing all of the information contained in the reference but this is not so great at ensuring our model is not just pushing out a huge number of words to game the recall score. To avoid this we use the **precision** metric which is calculated in almost the exact same way, but rather than dividing by the reference n-gram count, we divide by the model n-gram count.

  $$\frac{\textit{number of n-grams found in model and reference}}{\textit{number of n-grams in model}}$$

  Now that we both the recall and precision values, we can use them to calculate our **ROUGE F1 score** like so:

  $$2 * \frac{precision * recall}{precision + recall}$$

[1]n-gram is simply a grouping of tokens/words. A unigram (1-gram) would consist of a single word. A bigram (2-gram) consists of two consecutive words

That gives us a reliable measure of our model performance that relies not only on the model capturing as many words as possible (recall) but doing so without outputting irrelevant words (precision).

- *ROUGE-L*, it measures the longest common subsequence (LCS) between our model output and reference. The idea here is that a longer shared sequence would indicate more similarity between the two sequences. We can apply our recall and precision calculations just like before. In this project we also used the **Rouge-L-sum** metric which also measure the longest common subsequence, but newlines in the text are interpreted as sentence boundaries, so the sequence of n-grams is computed inside these sentences.

ROUGE is a great evaluation metric but comes with some drawbacks. In particular, ROUGE does not cater for different words that have the same meaning as it measures syntactical matches rather than semantics. So, if we had two sequences that had the same meaning, but used different words to express that meaning they could be assigned a low ROUGE score.

Due to this reason we should always consider human judgments as well.

8 Models comparison

We have tried three different models which are commonly used in the summarization task, in order to verify which of them is the best for our application.

8.1 BART

BART combines the pretraining procedures of BERT and GPT within the encoder-decoder architecture. The input sequences undergo one of several possible transformations, from simple masking to sentence permutation, token deletion, and document rotation. These modified inputs are passed through the encoder, and the decoder has to reconstruct the original texts. This makes the model more flexible as it is possible to use it for natural language understanding as well as natural language generation tasks, and it achieves state-of-the-art-performance on both.

8.2 PEGASUS

Like BART, PEGASUS is an encoder-decoder transformer. its pretraining objective is to predict masked sentences in multisentence texts. The authors argue that the closer the pretraining objective is to the downstream task, the more effective it is. With the aim of finding a pretraining objective that is closer to summarization than general language modeling, they automatically identified, in a very large corpus, sentences containing most of the content of their surrounding paragraphs (using summarization evaluation metrics as a heuristic for content overlap) and pretrained the PEGASUS model to reconstruct these sentences, thereby obtaining a state-of-the-art model for text summarization.

8.3 T5

The T5 model unifies all natural language understanding and natural language generation tasks by converting them into text-to-text tasks. All tasks are framed as sequence-to-sequence tasks, where adopting an encoder-decoder architecture is natural. The T5 architecture uses the original Transformer architecture.

8.4 Results

In this section we compared the performance evaluated with ROUGE metric of the various models. In the first table we can see the result obtained using the pretrained model.

Summarizer Models Comparison				
Models	**rouge1**	**rouge2**	**rougeL**	**rougeLsum**
BART-cnn	0.203312	0.040382	0.124216	0.124753
T5-base	0.202367	0.039603	0.131452	0.131624
pegasus-cnn_dailymail	0.200299	0.036340	0.127049	0.126664

In the second table we evaluate our finetuned model, we can se how using a domain specific dataset the performance on summarization task doubled, based on this results we decide to use BART-cnn-tos.

Summarizer Fine Tuned Models Comparison				
Models	**rouge1**	**rouge2**	**rougeL**	**rougeLsum**
BART-cnn-tos	0.492817	0.355159	0.388946	0.389677
T5-base-tos	0.380697	0.230032	0.274808	0.274247
pegasus-tos	0.411505	0.273133	0.310424	0.309713

Below is an example of abstractive summarization computed with BART-cnn-tos on the Amazon's terms of service :

- *This service provides archives of their terms of service so that changes can be viewed over time.*
- *You are responsible for maintaining the security of your Amazon account and for the activities on your account.*
- *Users who have been permanently banned from this service are not allowed to re-register under a new account.*
- *You must create an Amazon account to use this service.*
- *Voice data is collected and stored in a cloud-based service.*
- *Users should revisit the terms periodically, although in case of material changes, the service will notify.*
- *Instead of asking directly, this Service will assume your consent merely from your usage.*
- *The service can suspend your account for several reasons.*
- *Failure to enforce any provision of the Terms of Service does not constitute a waiver of such provision.*
- *This service does not guarantee that it or the products obtained through it meet your expectations or requirements.*
- *The service provider makes no warranty regarding uninterrupted, timely, secure or error-free service.*
- *Any liability on behalf of the service is only limited to $50.00.*
- *A complaint mechanism is provided for the handling of personal data.*

Today the limitation of these summarization models is that they have a maximum input length in terms of context tokens. Unfortunately, there is no single strategy to solve this problem, and it's an open and active research question.

We adopted the solution to split the text in blocks smaller than the maximum number of tokens accepted as input by the model, being careful not to cut off sentences, and apply recursively the computation of summaries.

9 Conclusions

In a good strategy analysis the firm needs to recognize and address the concerns of stakeholders and ask itself what responsibilities it has towards them. Ethical responsibility can play a crucial role. Even if the legal responsibilities often define only the minimum acceptable standards of firm behaviour, many times it is necessary to go further. The letter of the law cannot process or anticipate every possible business situation and new concerns such as internet privacy, data usage and artificial intelligence. The companies that are commited in being able to be clear and transparent towards customers or help them to be aware of the contracts that they are accepting need to address the *click-to-agree* problem. This plays a crucial role in the trust that customer have towards the firm and may also significantly increase their loyalty.

The service that we developed is able to summarize terms of service contracts and it also shows on screen the most important information as a list of bullet points. In this way, the user has a clear and simple view of all the clauses and terms that is about to accept.

Regarding the possible targets, a service with these features can be offered to all the companies that want to take care of their customers' trust: we believe that it is in the interest of the company itself to provide the user with a third-party mechanism (our service) to summarize and extract key information from their contracts; since the service is not owned by the company, but it is paying for it, the user can be sure that the summarization is legit and it has not been tampered with.

We thought that our service may be employed in a similar way as a *CAPTCHA*: the company can easily embed it in its web pages, in particular in contract boxes, in this way the user is forced to read the summarization in keypoints of the contract before continuing. In figure 13 is shown a mockup of the previously described application of the service.

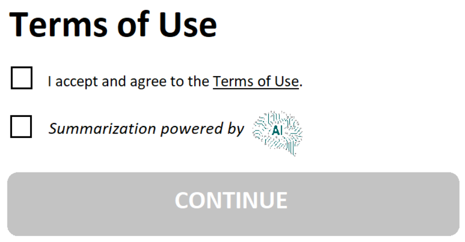


Figure 13: Mockup of the service.

This solution can also be adopted by all those companies that operate in the online industry and are configured as a hub of third-party services, in which users must accept new terms of service each time they purchase a service. To prove that they are concerned about the user at the time of purchase, in this way a firm can go beyond what is required by law, offering its customers a more transparent agreement, making them aware of their rights.

10 Deployment on Hugging face Space

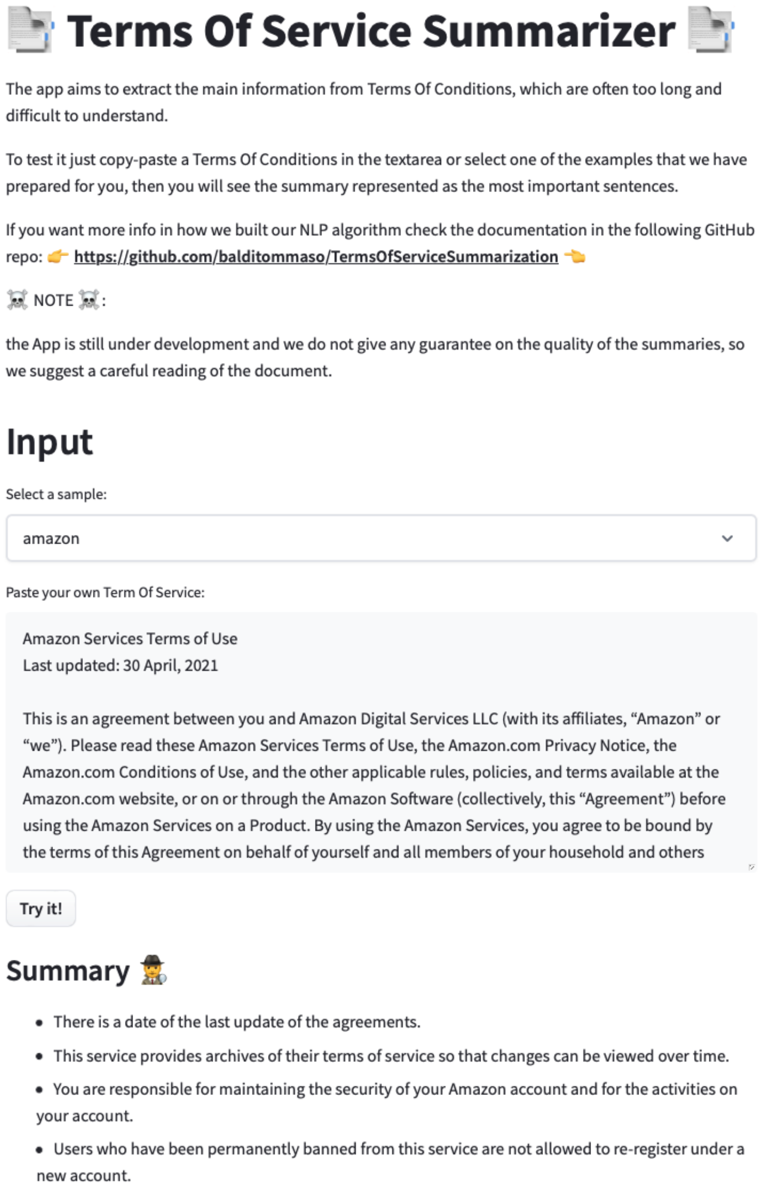
Hugging Face Spaces make it easy for you to create and deploy ML-powered demos in minutes. Under the hood, Spaces stores your code inside a git repository, just like the model and dataset repositories. Thanks to this, the same tools we use for all the other repositories on the Hub (git and git-lfs) also work for Spaces. Follow the same flow as in Getting Started with Repositories to add files to your Space. Each time a new commit is pushed, the Space will automatically rebuild and restart. Each Spaces environment is limited to 16GB RAM and 8 CPU cores.

The demo of our project has been built with the Streamlit Library, which allow us to turns pure Python scripts into shareable web apps.

The web interface is really easy, it is composed by a dropdown menu where the user can select one of the samples of Term Of Service document that we provide, otherwise the user can paste the target text in a textarea. When the text is ready the web app use the our model, also stored on Hugging Face, to summarize the given text and when the summaries are ready they will be displayed as list of short sentences.

The summary can take a while due to the computational capabilities of the free version of Hugging Face spaces, which are limited, and the model takes no more than 1024 tokens as input, then in low to document length this is broken into text blocks which are summarized one by one.

The demo is available at the following link: **Application Demo**



Terms Of Service Summarizer

The app aims to extract the main information from Terms Of Conditions, which are often too long and difficult to understand.

To test it just copy-paste a Terms Of Conditions in the textarea or select one of the examples that we have prepared for you, then you will see the summary represented as the most important sentences.

If you want more info in how we built our NLP algorithm check the documentation in the following GitHub repo: 👉 https://github.com/balditommaso/TermsOfServiceSummarization 👈

☠️ NOTE ☠️:

the App is still under development and we do not give any guarantee on the quality of the summaries, so we suggest a careful reading of the document.

Input

Select a sample:

amazon

Paste your own Term Of Service:

Amazon Services Terms of Use
Last updated: 30 April, 2021

This is an agreement between you and Amazon Digital Services LLC (with its affiliates, “Amazon” or “we”). Please read these Amazon Services Terms of Use, the Amazon.com Privacy Notice, the Amazon.com Conditions of Use, and the other applicable rules, policies, and terms available at the Amazon.com website, or on or through the Amazon Software (collectively, this “Agreement”) before using the Amazon Services on a Product. By using the Amazon Services, you agree to be bound by the terms of this Agreement on behalf of yourself and all members of your household and others

Summary 🕵️

- There is a date of the last update of the agreements.
- This service provides archives of their terms of service so that changes can be viewed over time.
- You are responsible for maintaining the security of your Amazon account and for the activities on your account.
- Users who have been permanently banned from this service are not allowed to re-register under a new account.

Figure 14: Demo of the application.

11 References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. Attention is all you need. arXiv:1706.03762, 2017.

[2] Lewis Tunstall, Leandro von Werra, and Thomas Wolf. Natural Language Processing with Transformers.

[3] Jay Alammar. The Illustrated Trans- former. http://jalammar.github.io/illustrated-transformer/, 2018.