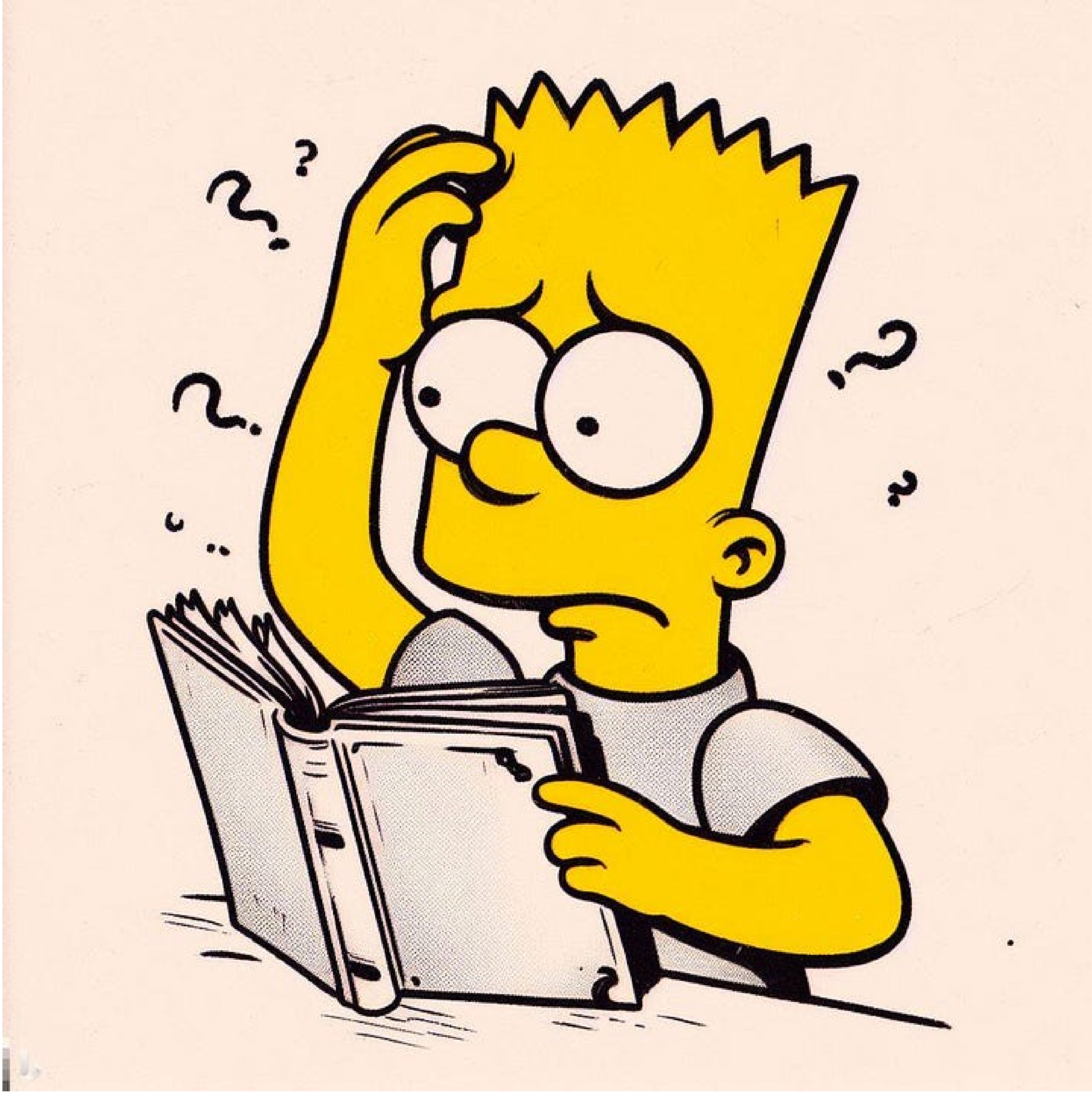
Text Summarization with Large Language Models

A Study on How to Harness the Power of LLMs for Natural Language Processing.

Luís Fernando Torres · Follow

27 min read · Just now



Bart Simpson Art Generated with Bing Dall-E 3

> **Note:** This article is an excerpt of my Kaggle Notebook, 📝 **Text Summarization with Large Language Models**. By clicking on the link, you will be able to read the full process step-by-step, as well as interact with the plots. Thank you!

Introduction

November 30th, 2022, marks a significant chapter in the History of **machine learning**. It was the day OpenAI released ChatGPT, setting a new benchmark for chatbots powered by **Large Language Models** and offering the public an unparalleled conversational experience.

Ever since then, large language models — also referred to as **LLMs** — , have been in the public eye due to the extensive number of tasks they are able to perform.

Examples include:

• **Text Summarization:** These models are able to perform a summarization of large texts, including legal texts, reviews, dialogues, among many others.

• **Sentiment Analysis:** They can read through reviews of products and services and classify them as positive, negative, or neutral. These can also be used in Finance to see if the general public feels *Bullish* or *Bearish* on certain securities.

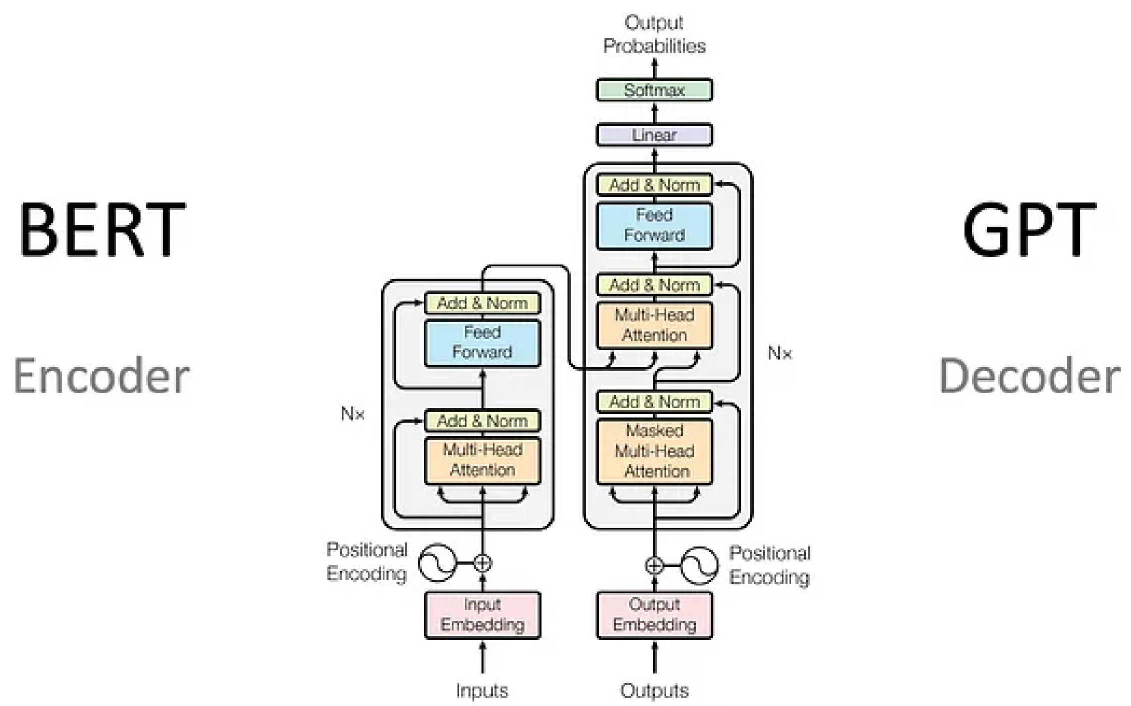
• **Language Translation:** They can provide real-time translations from one language to another.

• **Text-based Recommender Systems:** They can also recommend new products for a client based on their reviews on previously bought products.

But how do these models actually work? 🤔

The Transformer Architecture

To understand the current state of LLMs, we must go back to Google's 2017 **Attention is All You Need**. In this paper, the **Transformer** architecture was introduced to the world, and it changed the industry forever.



Foundation Models, Transformers, BERT and GPT

While recurrent neural networks could be used to enable computers to comprehend text, these models were extremely limited due to the fact that they only allowed the machine to process one word at a time, which would result in the model not being able to acquire the full context of a text.

The **transformer architecture**, however, is based on the attention mechanism, which allows the model to process an entire sentence or paragraph at once, rather than each word at a time. This is the main secret behind the possibility of full context comprehension, which gives much more power to all these language processing models.

The processing of text input with the transformer architecture is based on **tokenization**, which is the process of transforming words — as well as subwords — into tokens. These tokens are then mapped to dense **embeddings** — high-dimensional vectors — that contain **numerical tokens** that are able to capture the meaning of the original words, and serve as input for the transformer model.

Take the following example:

Original Text	Tokenized Text	Embedding of the Text
As she said this, she looked down at her hands, and was surprised to find that she had put on one of the rabbit's little gloves while she was talking.	['As', ' she', ' said', ' this', ',', ' she', ' looked', ' down', ' at', ' her', ' hands', ',', ' and', ' was', ' surprised', ' to', ' find', ' that', ' she', ' had', ' put', ' on', ' one', ' of', ' the', ' rabbit', "'s", ' little', ' gloves', ' while', ' she', ' was', ' talking', '.']	[2.49 0.22 -0.36 ... 0.27 0.48 -2.34]

By using this vector as input, the transformer model learns how to generate outputs based on the **probabilities of subsequent words that may naturally follow an input word.** This process gets repeated until the model creates an entire paragraph starting from an initial statement.

There is a very intriguing post on Andrej Karpathy's blog, ***The Unreasonable Effectiveness of Recurrent Neural Networks***, that explains why neural networks-based models are effective in predicting the next word of a text. One factor contributing to their effectiveness is the inherent *rules* in human languages, such as grammar, which constrain word usage in sentences.

When you feed your model with examples of written language — news articles, Twitter/X posts, product reviews, messages, dialogues, etc. — it implicitly acquires the rules of language through these examples, which helps it to predict sequences of words and generate human-like texts.

A large language model — such as *GPT*, *BERT*, *RoBERTa*, etc. — is a transformer model on a much larger scale. These models are built on an enormous amount of texts, so they learn and become experts in patterns and structures of language. The GPT-4, which is the model behind the premium version of ChatGPT, was trained on massive amounts of text data from the internet, such as books, articles, websites, etc.

It is also relevant to note that different languages exhibit different patterns and structures. While Western European languages like English, French, German, Spanish, Portuguese, and Italian may share many structural similarities, other languages, such as Arabic and Japanese, are very distinct, posing unique challenges to modeling.

This Notebook

The goal of this notebook is to demonstrate how Large Language Models can be used for several tasks related to language processing. In this case, I am going to leverage the power of **transfer learning** to build a model capable of summarizing dialogues.

For those of you who may not be aware, transfer learning is a machine learning technique in which we use a *pre-trained model* — that is already knowledgeable in a wide domain — and tailor its expertise for a specific task by training it in a specific dataset we might have. This process may also be referred to as **fine-tuning**.

The 🤗 Transformers library — which is one of the most popular libraries for working with deep learning tasks — offers the possibility of working with the following architectures:

Model Architectures
BART, BigBird-Pegasus, Blenderbot, BlenderbotSmall, Encoder decoder, FairSeq Machine-Translation, GPTSAN-japanese, LED, LongT5, M2M100, Marian, mBART, MT5, MVP, NLLB, NLLB-MOE, Pegasus, PEGASUS-X, PLBart, ProphetNet, SwitchTransformers, T5, UMT5, XLM-ProphetNet

The 🤗 **Transformers** library allows us to easily download and fine-tune state-of-the-art pre-trained models, and also allows us to easily work with both **TensorFlow** and **PyTorch** for several tasks related to Natural Language Preprocessing, Computer Vision, Audio, etc.

The Task

As previously mentioned, the task at hand is **Text Summarization**. From the documentation of the 🤗 Transformers library, summarization can be described as the creation of *a shorter version of a document or an article that captures all the important information*.

In this case, we are going to summarize dialogues by using a dataset containing chat texts.

The Dataset

For this task, we are going to use the **SamSum Dataset**, which contains three *csv* files for training, testing, and validation. All these files are structured into a specific `id`, a `dialogue`, and a `summary`. The SamSum dataset consists of chat texts, which is ideal for the summarization of dialogues.

The Model

As previously mentioned, we are going to harness the power of a pre-trained model for this task. In this case, I have decided to use the **BART** architecture, proposed in the 2019 paper BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. More specifically, I am going to fine-tune a version of BART that has been already trained to perform text summarization of news articles, which is the **facebook/bart-large-xsum** version.

Briefly explaining, BART is a denoising autoencoder that employs the strategy of distorting the input text in many ways, such as blanking out some words and flipping them around, and then learning to reconstruct it. BART has outperformed established models like RoBERTa and BERT on multiple NLP benchmarks, and it is especially efficient in summarization tasks, due to its ability to generate text and learn the context of the input text.

For a deeper comprehension of BART, I highly suggest you read the research paper linked above, where this architecture was first introduced.

Evaluation Metrics

Evaluating performance for language models can be quite tricky, especially when it comes to text summarization. The goal of our model is to produce a short sentence describing the content of a dialogue while maintaining all the important information within that dialogue.

One of the quantitative metrics we can employ to evaluate performance is the **ROUGE Score**. It is considered one of the best metrics for text summarization and it evaluates performance by comparing the quality of a machine-generated summary to a human-generated summary used for reference.

The similarities between both summaries are measured by analyzing the overlapping *n-grams*, either single words or sequences of words that are present in both summaries. These can be unigrams (ROUGE-1), where only the overlap of sole words is measured; bigrams (ROUGE-2), where we measure the overlap of two-word sequences; trigrams (ROUGE-3), where we measure the overlap of three-word sequences; etc. Besides that, we also have:

• **ROUGE-L**: It measures the *Longest Common Subsequence (LCS)* between the two summaries, which helps to capture content coverage of the machine-generated text. If both summaries have the sequence *"the apple is green"*, we have a match regardless of where they appear in both texts.

• **ROUGE-S**: It evaluates the overlap of skip-bigrams, which are bigrams that permit gaps between words. This helps to measure the coherence of a machine-generated summary. For example, in the phrase *"this apple is absolutely green"*, we find a match for the terms such as *"apple"* and *"green"*, if that is what we are looking for.

These scores might typically range from 0 to 100, where 0 indicates no match and 100 indicates a perfect match between both summaries.

Besides quantitative metrics, it is useful to use **human evaluation** to analyze the output of language models, since we are able to comprehend text in a way that a machine does not. So we might read the dialogue and then read the summary to check if the summarization is accurate or not.

Exploring the Dataset

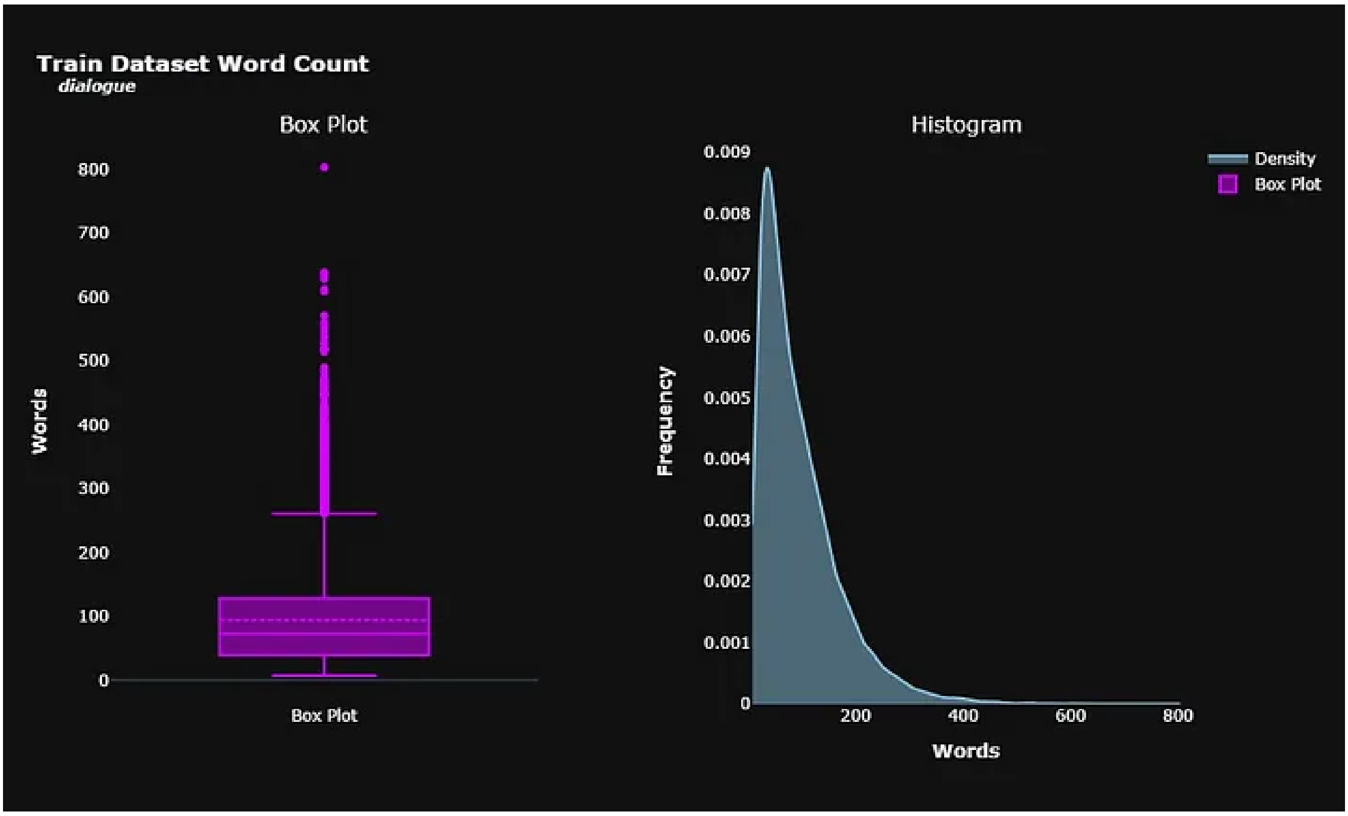
We can start our analysis of the dataset by loading all the three sets available, `train`, `test`, and `val`.

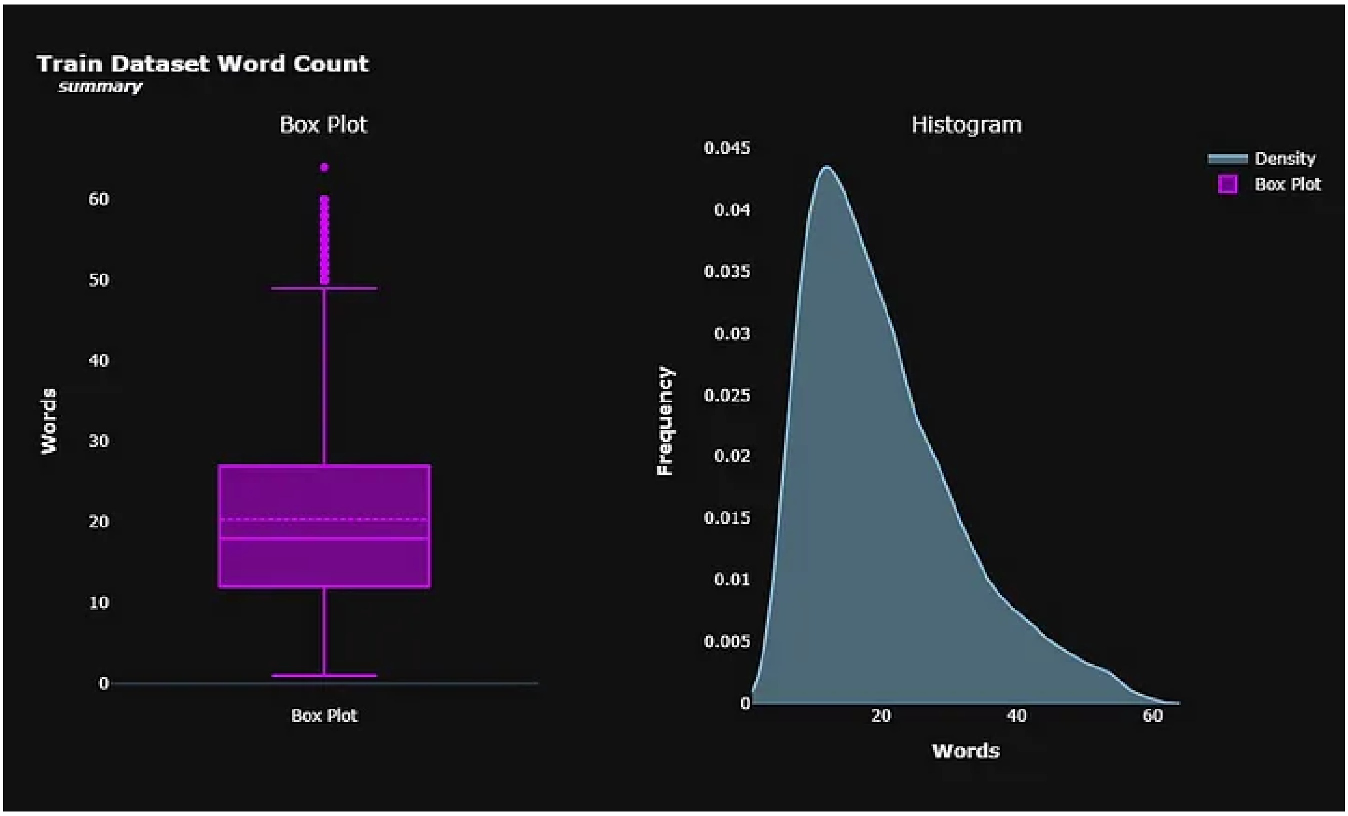
```
# Loading data
train = pd.read_csv('/kaggle/input/samsum-dataset-text-summarization/samsum-tra
test = pd.read_csv('/kaggle/input/samsum-dataset-text-summarization/samsum-test
val = pd.read_csv('/kaggle/input/samsum-dataset-text-summarization/samsum-valic
```

I am now going to analyze each dataset separately.

Train Dataset

We can analyze the length of both dialogues and summaries by counting the words in them. This might give us a clue on how these texts are structured.





On average, dialogues consist of about 94 words. We do have some outliers with very extensive texts, going way over 300 words per dialogue.

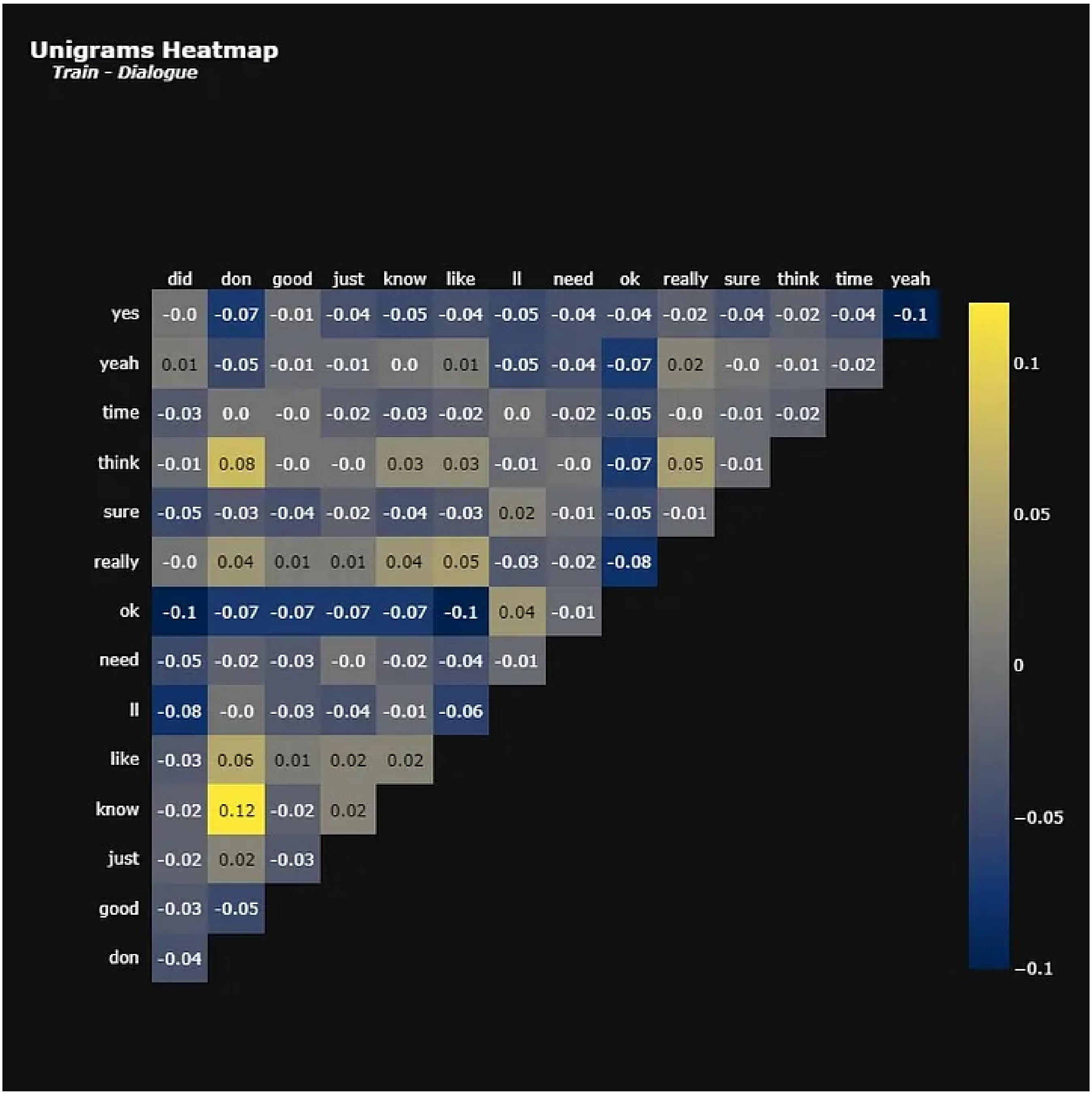
Summaries are naturally shorter texts, consisting of about 20 words on average, although we also have some outliers with extensive summaries.

We can also use scikit-learn's TfidfVectorizer to extract more info on the dialogues and summaries available. This function will give us a dataframe with the top *n* most frequent terms in the corpus, which we select by using the max_features parameter.

In this dataframe, each column represents the *n* most frequent terms in the overall corpus, while each row represents one entry in the original dataframe, such as train. For each term in each entry, we will see the TF-IDF score associated with it, which quantifies the relevance of a term in a given dialogue — or summary — relative to its frequency across all other dialogues — or summaries.

We will also use the ngram_range parameter to select the most frequent words (unigrams), the most frequent sequence of two words (bigrams), and the most frequent sequence of three words (trigrams). The stop_words = 'english' parameter will help us filter out common stop-words of the English language, which are words that do not add up much to the overall context, such as *"and"*, *"of"*, etc.

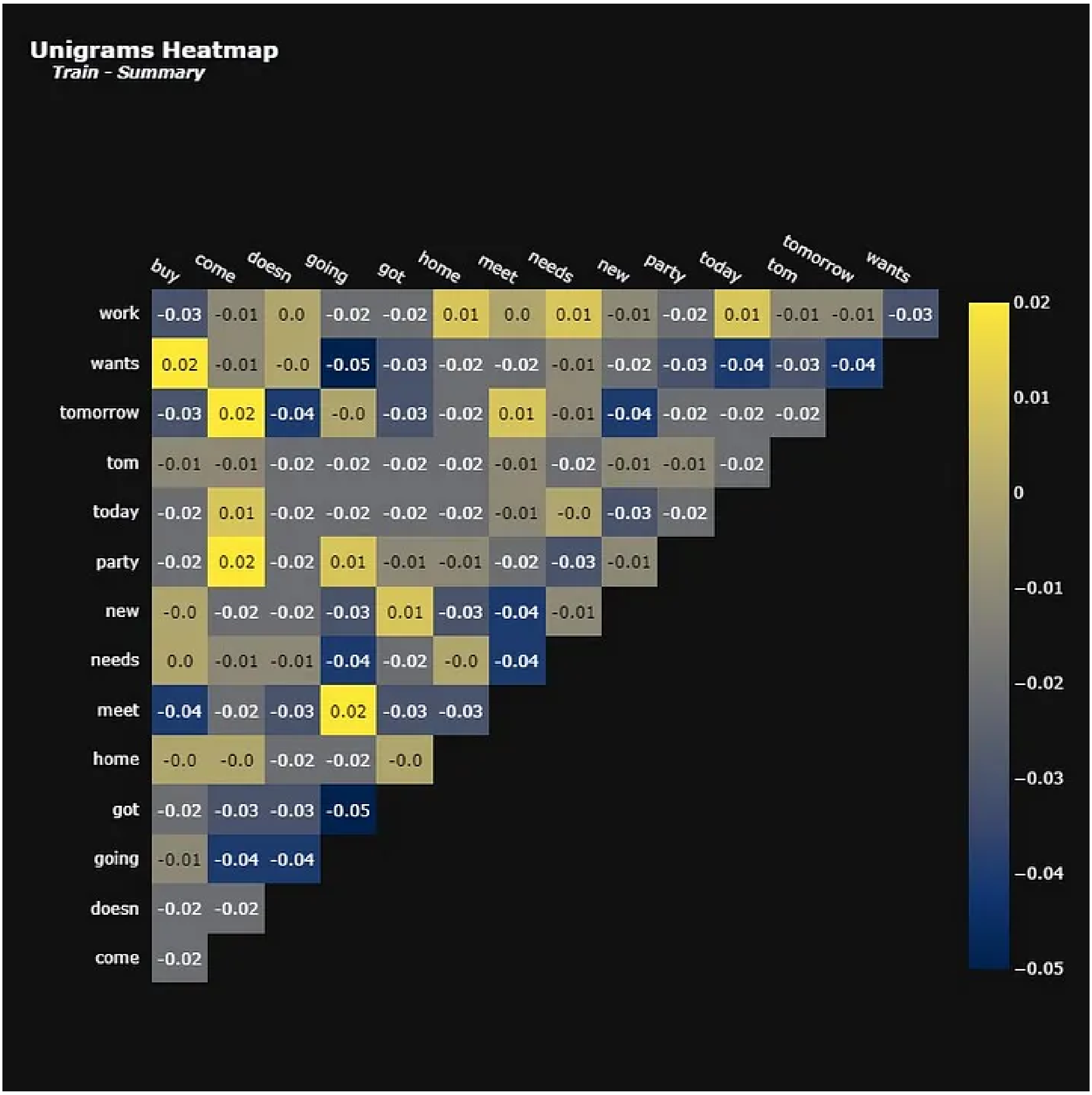
After measuring the most frequent terms, I will plot a heatmap displaying the correlations between these terms. This may help us understand how frequently they are used together in dialogues. For instance, how frequent is the occurrence of the word *"will"* when the word *"we"* is present?



You can see that the correlations between these terms are neither strongly positive nor strongly negative. The most positively correlated terms are *"don"* and *"know"*, at 0.12. It is relevant to observe that the TfidfVectorizer function performs some changes to the text, such as removing contractions, which explains why the word *don't* appears without its apostrophe *'t*.

It is also interesting to notice a negative correlation — although still not extremely significant — between the terms *"yes"* and *"yeah"*. Maybe this happens because it would be redundant to include both in the same dialogue, or perhaps the data captures a tendency of individuals to use *"yeah"* instead of *"yes"* during conversations. These are some hypotheses we can consider when analyzing this type of heatmaps.

Let's perform the same analysis to the summaries.

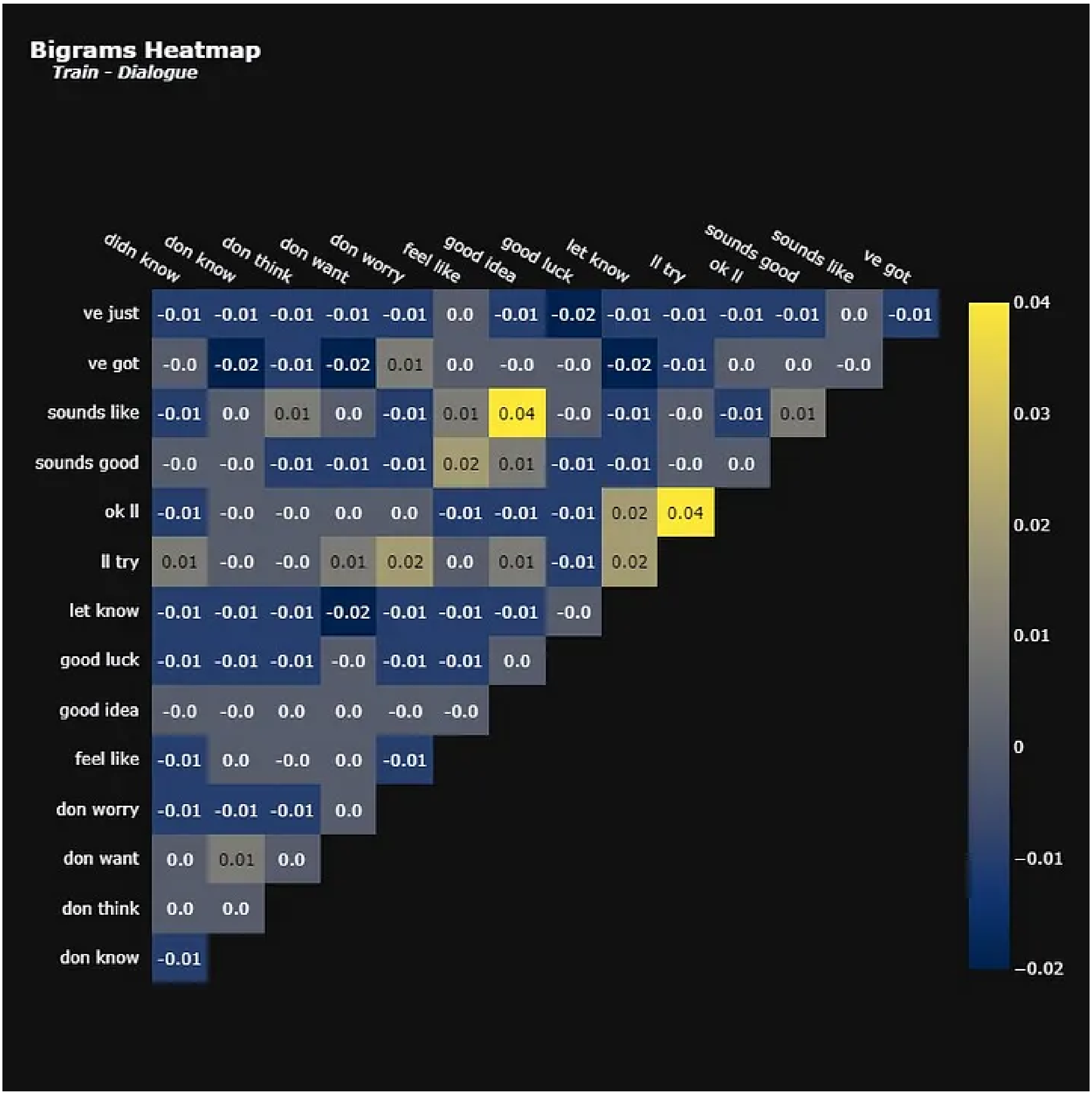


The correlations of terms in summaries seem to be more pronounced than those in dialogues, even though these correlations are still not strong. This suggests that summaries may convey relevant information more succinctly than full dialogues, which is exactly the idea behind a summary.

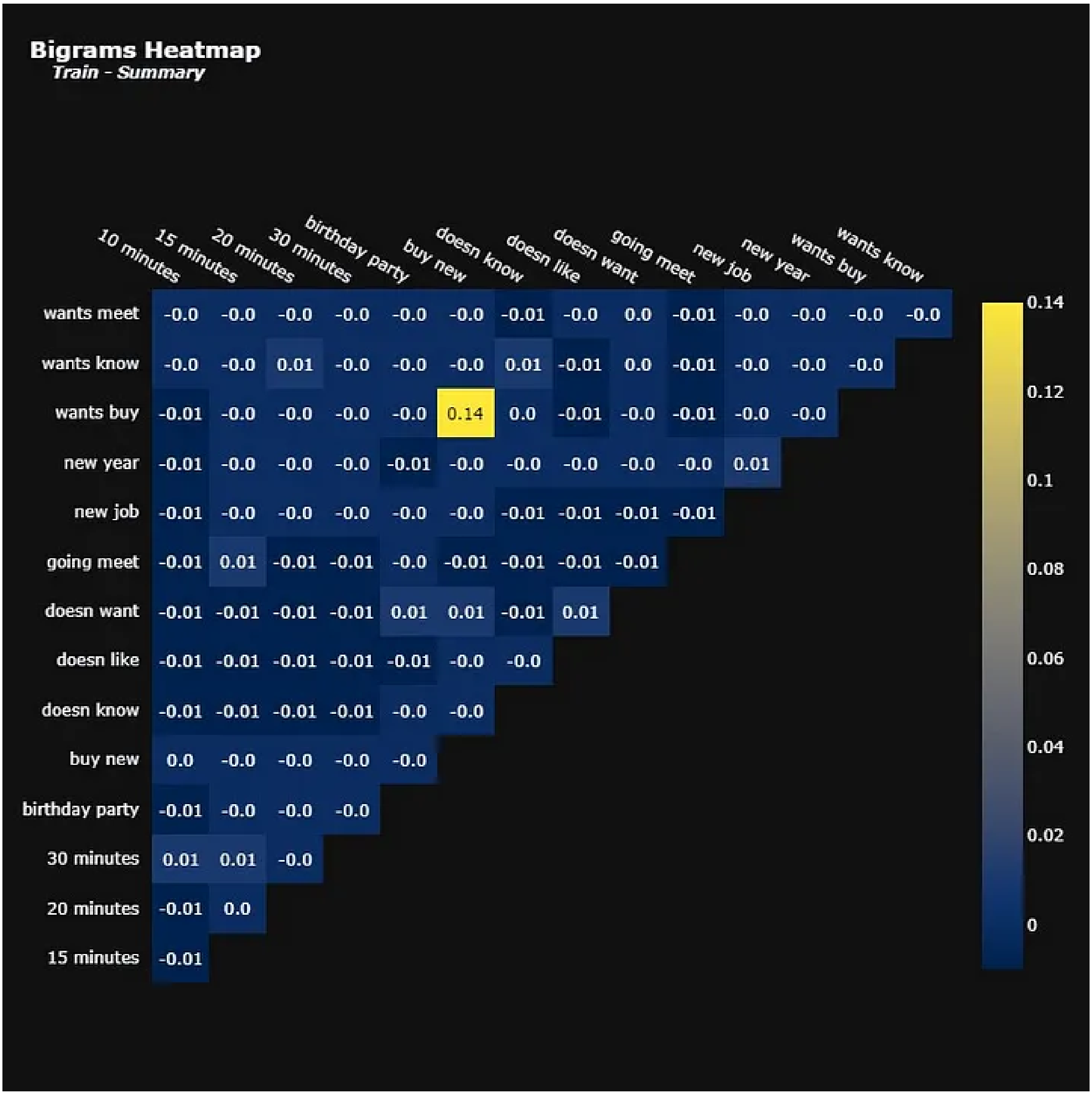
We have positively correlated pairs such as *"going"* and *"meet"*, *"come"* and *"party"*, as well as *"buy"* and *"wants"*. It makes perfect sense to see these unigrams appearing together across texts.

Conversely, it's reasonable for negatively correlated pairs **not** to co-occur frequently in texts, such as *"going"* and *"wants"*, and *"going"* and *"got"*.

Let's now analyze bigrams across dialogues and summaries.



Once more, the correlations are not extremely strong. Still, we can see some pairs that seem reasonable to be together, such as *"good idea"* and *"sounds like"*.



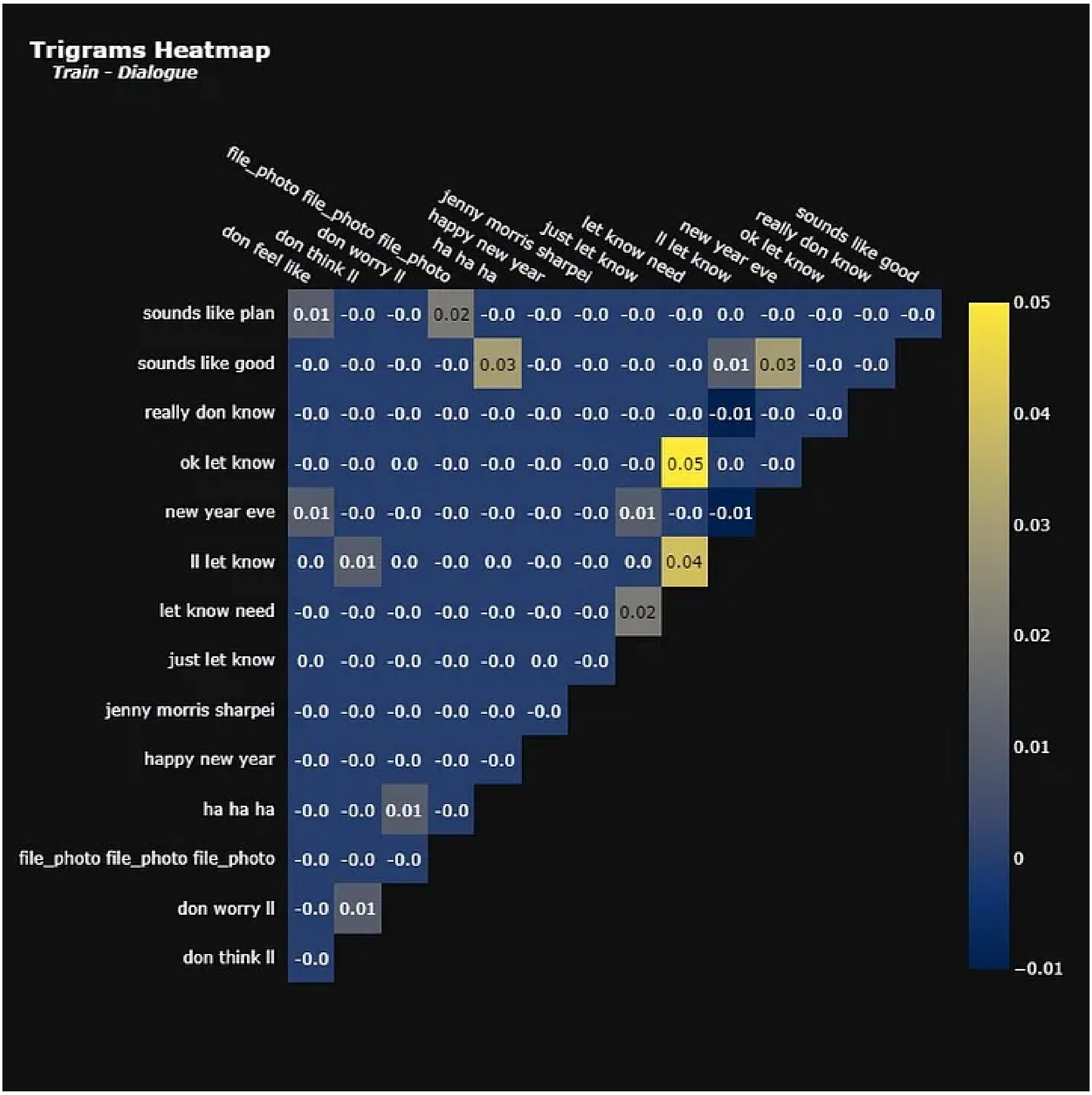
We have only one correlation between the pairs *"wants buy"* and *"buy new"*. The other terms do not appear to have any kind of correlation at all.

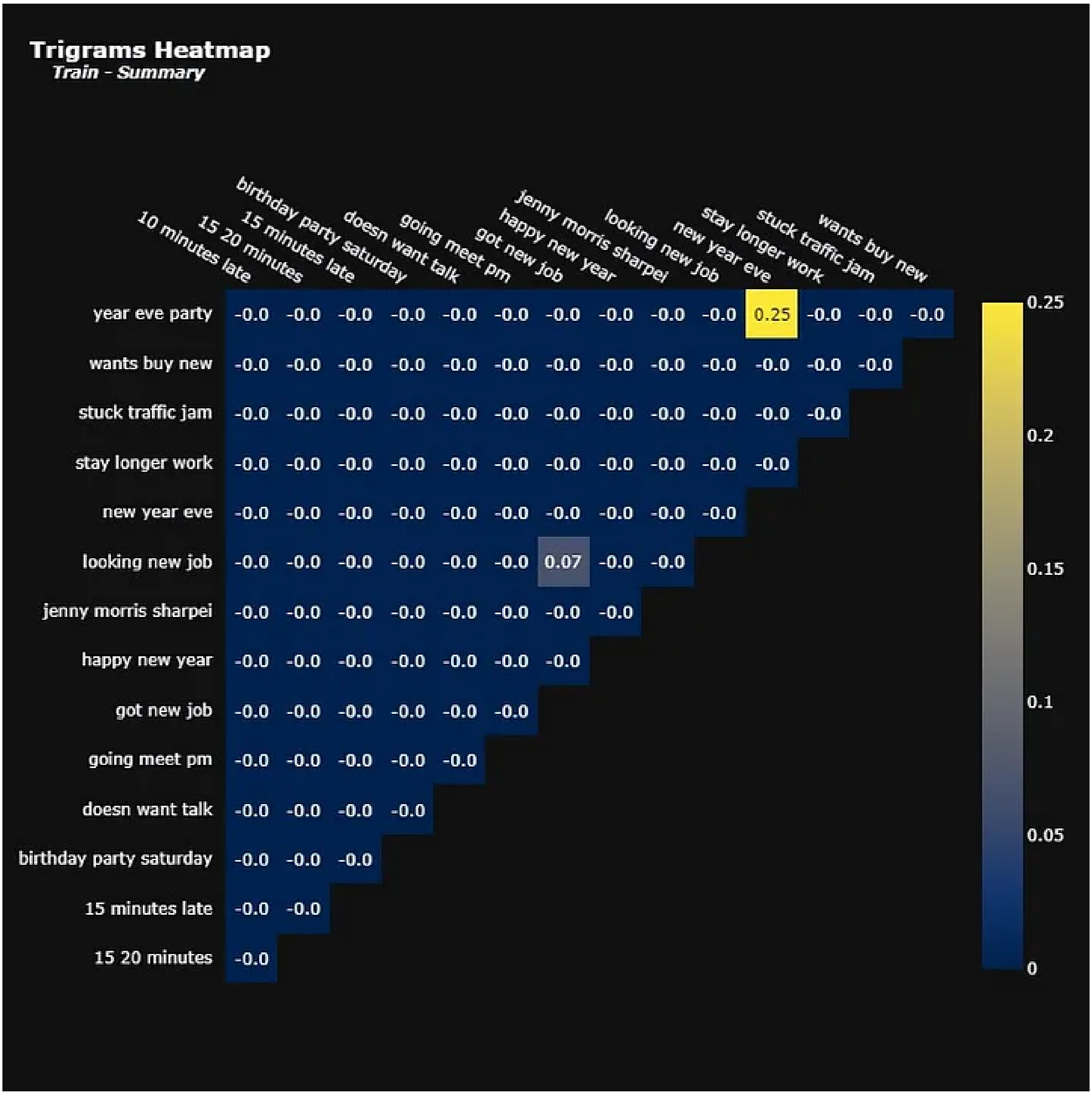
It is interesting to see the tendency of the summaries to contain information on minutes, which does not seem to be present in the dialogues. We can even investigate further this relationship by querying some summaries where the bigram *15 minutes* appears in the summary.

```
# Filtering dataset to see those containing the term '15 minutes' in the summar
filtered_train = train[train['summary'].str.contains('15 minutes', case=False,
filtered_train.head()
```

	id	dialogue	summary
136	13827893	Kate: I'm here <file_other>\r\nKate: there was no place in Red Lion\r\nSteven: hey! but it's quite far away\r\nKate: c'mon it's just 10 min by bike!\r\nSteven: yes, but I'm not by bike\r\nKate: car?\r\nSteven: nope\r\nSteven: by foot :P :P \r\nSteven: anyway google maps says 15 min and I'm there:D\r\nKate: ok, w8in ^^	Kate will meet with Steven in 15 minutes.
428	13811484-1	Jenny: Let's go out to eat.\r\nLucy: That sounds like fun.\r\nJenny: Where do you wanna go?\r\nLucy: Let me think a minute.\r\nJenny: I feel like Chinese.\r\nLucy: That sounds yummy.\r\nJenny: I know a good Chinese restaurant.\r\nLucy: How far away is it?\r\nJenny: It's only 10 minutes from my place.\r\nLucy: Do we have to book a table?\r\nJenny: Oh, no. We can walk right in.\r\nLucy: Cool. Will be in 15 minute. I'm really hungry!	Jenny and Lucy are going to a Chinese restaurant to eat. They do not need to book a table. Lucy will be at Jenny's in 15 minutes.
570	13818296	Danielle: hey where RU?\r\nJuan: I told u I'd be late!\r\nDanielle: but it's been almost 45 mins!\r\nDanielle: <file_gif>\r\nJuan: I'll be there in 15 minutes\r\nJuan: <file_gif>	Juan is almost 45 minutes late. He'll be there in 15 minutes.
1213	13682296-1	John: I know you will be outraged but I like to provoke you :P\r\nTyre: What is it?\r\nJohn: I talked to our neighbour today and I am really starting to think that religious people are just stupid.\r\nTyre: Gosh. You know it's a stupid claim.\r\nJohn: I know that there are some clever, religious individuals. But statistically religious people are stupid.\r\nTyre: It's not true. There are stupid religious people and clever ones, just like atheists.\r\nJohn: But most of academics are not religious.\r\nTyre: How do you know it?\r\nJohn: Experience but also some data I've seen.\r\nTyre: It's just not true.\r\nJohn: They are mostly people believing in things that have nothing to do with logic or reason: miracles, ghosts, witchcraft, just as our neighbour.\r\nTyre: I think it's only one part of them. There are theologians, people who actually know a lot about philosophy, logic etc.\r\nJohn: Yes, there are also people doing "scientifically" tarot, horoscopes and astrology.\r\nTyre: You ca...	John and Tyre's neighbour stopped John in the staircase and talked about some miracles for 15 minutes. John thinks that religious people are stupid. Tyre disagrees with this generalization.
1812	13820691	Madge: are you alive? xD\r\nDorothy: i'm still drunk\r\nMadge: xDDDDDDDD jeeez\r\nFelicia: I don't know...how much did i drink?\r\nMadge: like 10 rounds\r\nFelicia: SHIT \r\nFelicia: you gotta be kidding me ahahaha xDDDDDDDDDDDDDDDDD\r\nDorothy: of course she is\r\nDorothy: it was at least 15\r\nFelicia: ;_______________;\r\nFelicia: was nice to meet you girls...shame on me as always\r\nDorothy: oh stop talking\r\nDorothy: just live the moment B-)\r\nFelicia: how am i supossed to live the moment if i don't remember the half o the night XD\r\nDorothy: well it happens :p \r\nMadge: we gotta repeat it, i had a lot of fun :D\r\nDorothy: i'm in. in 15min?\r\nFelicia: you're crazy ;-;	Dorothy is still intoxicated after at least 15 rounds of drink yesterday and can't remember much of what happened. She would like to meet her friends for a drink again in 15 minutes.

The last row gives us an idea of why we see so many terms related to minutes in summaries, but not in dialogues. In dialogues, people may write "15min" together or even other forms of it, such as "15m", whereas the summaries give us a patternized description, making it natural to be more prominent than other forms to describe time.


Trigrams Heatmap
Train - Dialogue
don feel like
don think ll
don worry ll
file_photo file_photo file_photo
ha ha ha
happy new year
jenny morris sharpei
just let know
let know need
ll let know
new year eve
ok let know
really don know
sounds like good
sounds like plan 0.01 -0.0 -0.0 0.02 -0.0 -0.0 -0.0 -0.0 -0.0 0.0 -0.0 -0.0 -0.0 -0.0
sounds like good -0.0 -0.0 -0.0 -0.0 0.03 -0.0 -0.0 -0.0 -0.0 0.01 0.03 -0.0 -0.0
really don know -0.0 -0.0 -0.0 -0.0 -0.0 -0.0 -0.0 -0.0 -0.0 -0.01 -0.0 -0.0
ok let know -0.0 -0.0 0.0 -0.0 -0.0 -0.0 -0.0 -0.0 0.05 0.0 -0.0
new year eve 0.01 -0.0 -0.0 -0.0 -0.0 -0.0 -0.0 0.01 -0.0 -0.01
ll let know 0.0 0.01 0.0 -0.0 0.0 -0.0 -0.0 0.0 0.04
let know need -0.0 -0.0 -0.0 -0.0 -0.0 -0.0 -0.0 0.02
just let know 0.0 -0.0 -0.0 -0.0 -0.0 0.0 -0.0
jenny morris sharpei -0.0 -0.0 -0.0 -0.0 -0.0 -0.0
happy new year -0.0 -0.0 -0.0 -0.0 -0.0
ha ha ha -0.0 -0.0 0.01 -0.0
file_photo file_photo file_photo -0.0 -0.0 -0.0
don worry ll -0.0 0.01
don think ll -0.0
0.05
0.04
0.03
0.02
0.01
0
−0.01



Once more, we can see that the terms are not strongly correlated. But still, it is possible to see pairs that seem logical to appear together in the corpus.

I will now perform the exact same analysis on the `test` and `val` datasets. We can expect the same behavior as the ones seen during the analysis of the training set, which is why I will refrain from commenting on the following plots to avoid redundancy. However, if something different appears, we will surely investigate further.

> ***Note:*** *To avoid an overly extensive article, let's move on straight to the preprocessing part. If you still wish to see the EDA on the test and validation sets, don't forget to see the* 📝 ***Text Summarization with Large Language Models*** *Kaggle Notebook.*

Preprocessing Data

One of the main advantages of working with pre-trained models, such as BART, is that these models are usually extremely robust and require very little data preprocessing.

While performing the EDA, I noticed that we have some tags in a few texts, such as file_photo. Let's take a look at a few examples.

```
Theresa: <file_photo>
Theresa: <file_photo>
Theresa: Hey Louise, how are u?
Theresa: This is my workplace, they always give us so much food here 😊
Theresa: Luckily they also offer us yoga classes, so all the food isn't much of
Louise: Hey!! 😳
Louise: Wow, that's awesome, seems great 😎 Haha
Louise: I'm good! Are you coming to visit Stockholm this summer? 😳
Theresa: I don't think so :/ I need to prepare for Uni.. I will probably attenc
Louise: Nice! Do you already know which classes you will attend?
Theresa: Yes, it will be psychology :) I want to complete a few modules that I
Louise: Very good! Is it at the Uni in Prague?
Theresa: No, it will be in my home town :)
Louise: I have so much work right now, but I will continue to work until the er
Theresa: You must send me some pictures, so I can see where you live :)
Louise: I will, and of my cat and dog too 🤗
Theresa: Yeeeesss pls :)))
Louise: 👌 👌
Theresa: 😺 💕
```

I am going to use the clean_tags function defined below to remove these tags from the texts, so we can make them cleaner.

```
def clean_tags(text):
    clean = re.compile('<.*?>') # Compiling tags
    clean = re.sub(clean, '', text) # Replacing tags text by an empty string

    # Removing empty dialogues
    clean = '\n'.join([line for line in clean.split('\n') if not re.match('.*:\

    return clean
```

```
test1 = clean_tags(train['dialogue'].iloc[14727]) # Applying function to exampl
print(test1)
```

```
Theresa: Hey Louise, how are u?
Theresa: This is my workplace, they always give us so much food here 😊
Theresa: Luckily they also offer us yoga classes, so all the food isn't much of
Louise: Hey!! 😳
Louise: Wow, that's awesome, seems great 😎 Haha
Louise: I'm good! Are you coming to visit Stockholm this summer? 😳
Theresa: I don't think so :/ I need to prepare for Uni.. I will probably attenc
Louise: Nice! Do you already know which classes you will attend?
Theresa: Yes, it will be psychology :) I want to complete a few modules that I
Louise: Very good! Is it at the Uni in Prague?
Theresa: No, it will be in my home town :)
Louise: I have so much work right now, but I will continue to work until the er
Theresa: You must send me some pictures, so I can see where you live :)
Louise: I will, and of my cat and dog too 🤗
Theresa: Yeeeesss pls :)))
Louise: 👌 👌
Theresa: 😺 💕
```

You can see that we have successfully removed the tags from the texts. I am now going to define the clean_df function, in which we will apply the clean_tags to the entire datasets.

```
# Defining function to clean every text in the dataset.
def clean_df(df, cols):
    for col in cols:
        df[col] = df[col].fillna('').apply(clean_tags)
    return df
```

```
# Cleaning texts in all datasets
train = clean_df(train,['dialogue', 'summary'])
test = clean_df(test,['dialogue', 'summary'])
val = clean_df(val,['dialogue', 'summary'])
```

The tags have been removed from the texts. It's beneficial to conduct such data cleansing to eliminate noise — information that might not significantly contribute to the overall context and could potentially impair performance.

I am now going to perform some preprocessing that is necessary to prepare our data to serve as input to the pre-trained model and for fine-tuning. Most of what I'm doing here is a part of the tutorial on Text Summarization described in the 🤗 Transformers documentation, which you can see here.

First, I am going to use the 🤗 Datasets library to convert our Pandas Dataframes to Datasets. This is going to make our data ready to be processed across the whole Hugging Face ecosystem.

```
# Transforming dataframes into datasets
train_ds = Dataset.from_pandas(train)
test_ds = Dataset.from_pandas(test)
val_ds = Dataset.from_pandas(val)

# Visualizing results
print(train_ds)
print('\n' * 2)
print(test_ds)
print('\n' * 2)
print(val_ds)
```

```
Dataset({
    features: ['id', 'dialogue', 'summary', '__index_level_0__'],
    num_rows: 14731
})



Dataset({
    features: ['id', 'dialogue', 'summary'],
    num_rows: 819
})



Dataset({
    features: ['id', 'dialogue', 'summary'],
```

```
    num_rows: 818
})
```

To see the content inside a 🤗 Dataset, we can select a specific row, as below.

```
train_ds[0] # Visualizing the first entry
```

```
{'id': '13818513',
 'dialogue': "Amanda: I baked  cookies. Do you want some?\r\nJerry: Sure!\r\nAm
 'summary': 'Amanda baked cookies and will bring Jerry some tomorrow.',
 '__index_level_0__': 0}
```

This way, we can see the original ID, the dialogue, as well as the reference summary. __index_level_0__ does not add anything to the data and will be removed further.

After successfully converting the pandas dataframes to 🤗 Datasets, we can move on to the modeling process.

Modeling

As I have previously mentioned, we are going to fine-tune a version of BART that has been trained on several news articles for text summarization, facebook/bart-large-xsum.

I will briefly demonstrate this model by loading a summarization pipeline with it to show you how it works on news data.

```
# Loading summarization pipeline with the bart-large-cnn model
summarizer = pipeline('summarization', model = 'facebook/bart-large-xsum')
```

As an example, I am going to use the following news article, published on CNN on October 24th, 2023, *Bobi, the world's oldest dog ever, dies aged 31*. Notice that this is a totally unseen news article that I'm passing to the model, so we can see how it performs.

```
news = '''Bobi, the world’s oldest dog ever, has died after reaching the
almost inconceivable age of 31 years and 165 days, said Guinness World Records
(GWR) on Monday.
His death at an animal hospital on Friday was initially announced by
veterinarian Dr. Karen Becker.
She wrote on Facebook that “despite outliving every dog in history,
his 11,478 days on earth would never be enough, for those who loved him.”
There were many secrets to Bobi’s extraordinary old age, his owner
Leonel Costa told GWR in February. He always roamed freely, without a leash
or chain, lived in a “calm, peaceful” environment and ate human food soaked
in water to remove seasonings, Costa said.
He spent his whole life in Conqueiros, a small Portuguese village about
150 kilometers (93 miles) north of the capital Lisbon, often wandering around
with cats.
Bobi was a purebred Rafeiro do Alentejo – a breed of livestock
guardian dog – according to his owner. Rafeiro do Alentejos have a
life expectancy of about 12-14 years, according to the American Kennel Club.
But Bobi lived more than twice as long as that life expectancy,
surpassing an almost century-old record to become the oldest living dog
and the oldest dog ever – a title which had previously been held by
Australian cattle-dog Bluey, who was born in 1910 and lived to be 29 years
and five months old.
However, Bobi’s story almost had a different ending.
When he and his three siblings were born in the family’s woodshed,
Costa’s father decided they already had too many animals at home.
Costa and his brothers thought their parents had taken all the puppies
away to be destroyed. However, a few sad days later, they found Bobi alive,
safely hidden in a pile of logs.
The children hid the puppy from their parents and, by the time
Bobi’s existence became known, he was too old to be put down and went on
to live his record-breaking life.
His 31st birthday party in May was attended by more than 100 people
and a performing dance troupe, GWR said.
His eyesight deteriorated and walking became harder as Bobi grew older
but he still spent time in the backyard with the cats, rested more and
napped by the fire.
“Bobi is special because looking at him is like remembering the people
who were part of our family and unfortunately are no longer here,
like my father, my brother, or my grandparents who have already left
this world,” Costa told GWR in May. “Bobi represents those generations.”
'''
summarizer(news) # Using the pipeline to generate a summary of the text above
```

```
[{'summary_text': 'The world’s oldest dog has died,
Guinness World Records has confirmed.'}]
```

You can observe that the model is able to accurately produce a much shorter text consisting of the most relevant information present in the input text. This is a successful summarization.

However, this model has been trained mainly on datasets consisting of several news articles from CNN and the Daily Mail, not on much dialogue data. This is why I'm going to fine-tune it with the SamSum dataset.

Let's go ahead and load BartTokenizer and BartForConditionalGeneration using the *facebook/bart-large-xsum* checkpoint.

```
checkpoint = 'facebook/bart-large-xsum' # Model
tokenizer = BartTokenizer.from_pretrained(checkpoint) # Loading Tokenizer
```

```
model = BartForConditionalGeneration.from_pretrained(checkpoint) # Loading Mode
```

We can also print below the architecture of the model.

```
BartForConditionalGeneration(
  (model): BartModel(
    (shared): Embedding(50264, 1024, padding_idx=1)
    (encoder): BartEncoder(
      (embed_tokens): Embedding(50264, 1024, padding_idx=1)
      (embed_positions): BartLearnedPositionalEmbedding(1026, 1024)
      (layers): ModuleList(
        (0-11): 12 x BartEncoderLayer(
          (self_attn): BartAttention(
            (k_proj): Linear(in_features=1024, out_features=1024,
bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024,
bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024,
bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024,
bias=True)
          )
          (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05,
elementwise_affine=True)
          (activation_fn): GELUActivation()
```

```
          (fc1): Linear(in_features=1024, out_features=4096, bias=True)
          (fc2): Linear(in_features=4096, out_features=1024, bias=True)
          (final_layer_norm): LayerNorm((1024,), eps=1e-05,
elementwise_affine=True)
        )
      )
      (layernorm_embedding): LayerNorm((1024,), eps=1e-05,
elementwise_affine=True)
    )
    (decoder): BartDecoder(
      (embed_tokens): Embedding(50264, 1024, padding_idx=1)
      (embed_positions): BartLearnedPositionalEmbedding(1026, 1024)
      (layers): ModuleList(
        (0-11): 12 x BartDecoderLayer(
          (self_attn): BartAttention(
            (k_proj): Linear(in_features=1024, out_features=1024,
bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024,
bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024,
bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024,
bias=True)
          )
          (activation_fn): GELUActivation()
          (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05,
elementwise_affine=True)
          (encoder_attn): BartAttention(
            (k_proj): Linear(in_features=1024, out_features=1024,
bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024,
bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024,
bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024,
bias=True)
          )
          (encoder_attn_layer_norm): LayerNorm((1024,), eps=1e-05,
elementwise_affine=True)
          (fc1): Linear(in_features=1024, out_features=4096, bias=True)
          (fc2): Linear(in_features=4096, out_features=1024, bias=True)
          (final_layer_norm): LayerNorm((1024,), eps=1e-05,
elementwise_affine=True)
        )
      )
      (layernorm_embedding): LayerNorm((1024,), eps=1e-05,
elementwise_affine=True)
    )
  )
  (lm_head): Linear(in_features=1024, out_features=50264, bias=False)
)
```

It is possible to see that the models consist of an encoder and a decoder, we can see the Linear Layers, as well as the activation functions, which use ***GeLU***, instead of the more typical ***ReLU***.

It is also interesting to observe the output layer, **lm_head**, which shows us that this model is ideal for generating outputs with a vocabulary size — out_features=50264 — this shows us that this architecture is adequate for summarization tasks, as well as other tasks, such as translation for example.

Now we must preprocess our datasets and use BartTokenizer so that our data is legible for the BART model.

The following preprocess_function can be directly copied from the 🤗 Transformers documentation, and it serves well to preprocess data for several NLP tasks. I am going to delve a bit deeper into how it preprocesses the data by explaining the steps taken.

- **inputs = [doc for doc in examples["dialogue"]]:** In this line, we are iterating over every dialogue in the dataset and saving them as input to the model.
- **model_inputs = tokenizer(inputs, max_length=1024, truncation=True):** Here, we are using the tokenizer to convert the input dialogues into tokens that can be easily understood by the BART model. The truncation=True parameter ensures that all dialogues have a maximum number of 1024 tokens, as defined by the max_length parameter.
- **labels = tokenizer(text_target=examples["summary"], max_length=128, truncation=True):** This line performs a very similar tokenization process as the one above. This time, however, it tokenizes the target variable, which is our summaries. Also, note that the max_length here is significantly lower, at 128. This implies that we expect summaries to be a much shorter text than that of dialogues.
- **model_inputs["labels"] = labels["input_ids"]:** This line is essentially adding the tokenized labels to the preprocessed dataset, alongside the tokenized inputs.

```
def preprocess_function(examples):
    inputs = [doc for doc in examples["dialogue"]]
    model_inputs = tokenizer(inputs, max_length=1024, truncation=True)
```

```
    # Setup the tokenizer for targets
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(examples["summary"], max_length=128, truncation=True

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs
```

```
# Applying preprocess_function to the datasets
tokenized_train = train_ds.map(preprocess_function, batched=True,
                               remove_columns=['id', 'dialogue', 'summary', '__

tokenized_test = test_ds.map(preprocess_function, batched=True,
                               remove_columns=['id', 'dialogue', 'summary']) #

tokenized_val = val_ds.map(preprocess_function, batched=True,
                               remove_columns=['id', 'dialogue', 'summary']) #

# Printing results
print('\n' * 3)
print('Preprocessed Training Dataset:\n')
print(tokenized_train)
print('\n' * 2)
print('Preprocessed Test Dataset:\n')
print(tokenized_test)
print('\n' * 2)
print('Preprocessed Validation Dataset:\n')
print(tokenized_val)
```

```
Preprocessed Training Dataset:

Dataset({
    features: ['input_ids', 'attention_mask', 'labels'],
    num_rows: 14731
})



Preprocessed Test Dataset:

Dataset({
    features: ['input_ids', 'attention_mask', 'labels'],
    num_rows: 819
})
```

```
Preprocessed Validation Dataset:

Dataset({
    features: ['input_ids', 'attention_mask', 'labels'],
    num_rows: 818
})
```

Our tokenized datasets consist now of only three features, input_ids, attention_mask, and labels. Let's print a sample from our tokenized train dataset to investigate further how the preprocess function altered the data.

```
nput_ids:
[0, 10127, 5219, 35, 38, 17241, 1437, 15269, 4, 1832, 47, 236, 103, 116,
50121, 50118, 39237, 35, 9136, 328, 50121, 50118, 10127, 5219, 35, 38,
581, 836, 47, 3859, 48433, 2]


attention_mask:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1]


sample:
[0, 10127, 5219, 17241, 15269, 8, 40, 836, 6509, 103, 3859, 4, 2]
```

Let's dive a deep further into what each feature means.

- **input_ids**: These are the token IDs mapped to the dialogues. Each token represents a word or subword that can be perfectly understood by the BART model. For instance, the number *5219* could be a map to a word like *"hello"* in BART's vocabulary. Each word has its unique token in this context.
- **attention_mask**: This mask indicates which tokens the model should pay attention to and which tokens should be ignored. This is often used in the context of padding — when some tokens are used to equalize the lengths of sentences — but most of these padding tokens do not hold any meaningful information, so the attention mask ensures the model does not focus on them. In the case of this specific sample, all tokens are masked as '1', meaning they are all relevant and none of them are used for padding.

- **labels**: Similarly to the first feature, these are tokenized word and subwords in the summaries. These are the tokens that the model will be trained on to give as output.

We must now use `DataCollatorForSeq2Seq` to batch the data. These data collators may also automatically apply some processing techniques, such as padding. They are important for the task of fine-tuning models and are also present in the 🤗 Transformers documentation for text summarization.

```
# Instantiating Data Collator
data_collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model=model)
```

Next, I am going to load the ROUGE metrics and define a new function to evaluate the model.
The compute_metrics function is also available in the documentation. In this function, we are extracting the model-generated summaries, as well as the human-generated summaries, and decoding them. We then use rouge to compare how similar they are to evaluate performance.

```
metric = load_metric('rouge') # Loading ROUGE Score
```

```
Downloading builder script:   0%|          | 0.00/2.16k [00:00<?, ?B/s]
```

```
def compute_metrics(eval_pred):
    predictions, labels = eval_pred# Obtaining predictions and true labels

    # Decoding predictions
    decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=Tru

    # Obtaining the true labels tokens, while eliminating any possible masked t
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)

    # Rouge expects a newline after each sentence
    decoded_preds = ["\n".join(nltk.sent_tokenize(pred.strip())) for pred in de
```

```
    decoded_labels = ["\n".join(nltk.sent_tokenize(label.strip())) for label ir


    # Computing rouge score
    result = metric.compute(predictions=decoded_preds, references=decoded_label
    result = {key: value.mid.fmeasure * 100 for key, value in result.items()} #

    # Add mean-generated length
    prediction_lens = [np.count_nonzero(pred != tokenizer.pad_token_id) for pre
    result["gen_len"] = np.mean(prediction_lens)

    return {k: round(v, 4) for k, v in result.items()}
```

We now use the Seq2SeqTrainingArguments class to set some relevant settings for fine-tuning. I will first define a directory to serve as output, and then define the evaluation strategy, learning rate, etc.

This class can be quite extensive, with several different parameters. I highly suggest you take your time with the documentation to get familiar with them.

```
# Defining parameters for training
'''
Please don't forget to check the documentation.
Both the Seq2SeqTrainingArguments and Seq2SeqTrainer classes have
quite an extensive list of parameters.

doc: https://huggingface.co/docs/transformers/v4.34.1/en/main_classes/trainer

'''
training_args = Seq2SeqTrainingArguments(
    output_dir = 'bart_samsum',
    evaluation_strategy = "epoch",
    save_strategy = 'epoch',
    load_best_model_at_end = True,
    metric_for_best_model = 'eval_loss',
    seed = seed,
    learning_rate=2e-5,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    gradient_accumulation_steps=2,
    weight_decay=0.01,
    save_total_limit=2,
    num_train_epochs=4,
    predict_with_generate=True,
    fp16=True,
```

```
        report_to="none"
    )
```

Finally, the Seq2SeqTrainer class allows us to use **PyTorch** to fine-tune the model. In this class, we are defining the model, the training arguments, the datasets used for training and evaluation, the tokenizer, the data_collator, and the metrics.

```
# Defining Trainer
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_test,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)
trainer.train() # Training model
```

[7364/7364 1:46:39, Epoch 3/4]

Epoch	Training Loss	Validation Loss	Rouge1	Rouge2	Rougel	Rougelsum	Gen Len
0	1.379400	1.484487	52.580300	27.111300	42.727600	48.367000	32.069600
2	1.074100	1.443861	52.815600	28.125900	43.714700	48.571200	29.147700
2	0.849400	1.525784	52.502500	27.853400	43.665300	48.442600	31.437100
3	0.681800	1.623999	52.693800	27.407100	43.325900	48.433200	30.183200

```
Out[69]:
TrainOutput(global_step=7364, training_loss=1.011966500186454, metrics={'train_ru
ntime': 6402.2537, 'train_samples_per_second': 9.204, 'train_steps_per_second':
1.15, 'total_flos': 3.4981815168344064e+16, 'train_loss': 1.011966500186454, 'epo
ch': 4.0})
```

Results of Training

We finally finished fine-tuning after 4 epochs. Since we had load_best_model_at_end = True in the training arguments, the Trainer automatically saves the model with the best performance, which in this case is the one with the lowest Validation Loss.

The second epoch was the one with the lowest validation loss, at **1.443861**. It also achieved the highest Rouge1 and Rouge2 scores, as well as the highest Rougelsum score.

I have not presented the **Rougelsum** score previously. According to the documentation of the rouge-score library, we can conclude that this is similar to the RougeL score, but it measures content coverage at a sentence-by-sentence level, instead of the entire summary.

The Gen Len column gives us the average length of the model-generated summaries. It is relevant to remember that we want short, yet informative, texts. In this case, the second epoch also yielded the shortest summaries on average.

Evaluating and Saving Model

After training and testing the model, we can evaluate its performance on the `validation` dataset. We can use the `evaluate` method for that.

```
# Evaluating model performance on the tokenized validation dataset
validation = trainer.evaluate(eval_dataset = tokenized_val)
print(validation) # Printing results
```

```
[205/205 04:42]

{'eval_loss': 1.4104626178741455, 'eval_rouge1': 53.8804, 'eval_rouge2': 29.2329,
'eval_rougeL': 44.774, 'eval_rougeLsum': 49.8255, 'eval_gen_len': 28.8839, 'eval_
runtime': 290.2637, 'eval_samples_per_second': 2.818, 'eval_steps_per_second': 0.
706, 'epoch': 4.0}
```

Results of the Validation

This outputs the same scores we have previously seen during training and testing. Here, we can notice that we have even **higher** performance in every metric compared to the performance in the testing set. When it comes to the `Gen Len`, we also have more concise summaries in the validation set.

Considering that our results seem to be satisfactory at this point, we can go ahead and use the `save_model` method to save our fine-tuned model in the `bart_finetuned_samsum` directory.

```
# Saving model to a custom directory
directory = "bart_finetuned_samsum"
trainer.save_model(directory)

# Saving model tokenizer
tokenizer.save_pretrained(directory)
```

```
('bart_finetuned_samsum/tokenizer_config.json',
 'bart_finetuned_samsum/special_tokens_map.json',
 'bart_finetuned_samsum/vocab.json',
 'bart_finetuned_samsum/merges.txt',
 'bart_finetuned_samsum/added_tokens.json')
```

After saving your model, you can easily upload it to Hugging Face Models and use it on new datasets and texts.

The fine-tuned model we trained here is now available for everyone on Hugging Face, and you can have access to it by clicking on luisotorres/bart-finetuned-samsum.

Let's load the model, using the summarization pipeline, and generate some summaries for human evaluation, where we evaluate if the model-generated summaries are accurate or not.

```
# Loading summarization pipeline and model
summarizer = pipeline('summarization', model = 'luisotorres/bart-
finetuned-samsum')
```

After loading the pipeline, we can now produce some summaries. I'll first start by using examples from the validation dataset, so we can compare our model-generated summaries to the reference summaries.

```
Original Dialogue:

John: doing anything special?
Alex: watching 'Millionaires' on tvn
Sam: me too! He has a chance to win a million!
John: ok, fingers crossed then! :)



Reference Summary:

Alex and Sam are watching Millionaires.



Model-generated Summary:

[{'summary_text': "Alex and Sam are watching 'Millionaires' on tvn."}]
```

The model-generated summary is just a bit longer than the reference summary, but it still captures quite well the content of the dialogue.

Let's see another example.

```
Original Dialogue:

Madison: Hello Lawrence are you through with the article?
Lawrence: Not yet sir.
Lawrence: But i will be in a few.
Madison: Okay. But make it quick.
Madison: The piece is needed by today
Lawrence: Sure thing
Lawrence: I will get back to you once i am through.



Reference Summary:

Lawrence will finish writing the article soon.



Model-generated Summary:
```

```
[{'summary_text': "Lawrence hasn't finished with the article yet,
but he will be in a few minutes. Madison needs the piece by today."}]
```

Once again, the model-generated summary is longer than the reference summary. However, I would definitely say that the model-generated summary is more informative than the reference one because it lets us know that there's a sense of urgency for Lawrence to finish the article, since Madison needs it by today.
Let's see another example.

```
Original Dialogue:

Robert: Hey give me the address of this music shop you mentioned before
Robert: I have to buy guitar cable
Fred: Catch it on google maps
Robert: thx m8
Fred: ur welcome



Reference Summary:

Robert wants Fred to send him the address of the music shop as he needs to
buy guitar cable.



Model-generated Summary:

[{'summary_text': 'Fred gives Robert the address of the music shop where
he will buy guitar cable.'}]
```

In this case, while the generated text captures the essence of the dialogue, it suffers from a lack of clarity due to ambiguity. Specifically, the pronoun *he* creates uncertainty about whether Fred or Robert intends to buy the guitar cable. In the original dialogue, it is clearly specified that it is Robert the one who has to buy the cable.

Now that we have been able to compare summaries, we can create some dialogues and input them into the model to check how it performs on them.

```
Original Dialogue:

John: Hey! I've been thinking about getting a PlayStation 5. Do you think
it
is worth it?
Dan: Idk man. R u sure ur going to have enough free time to play it?
John: Yeah, that's why I'm not sure if I should buy one or not. I've been
working so much lately idk if I'm gonna be able to play it as much as I'd
like.



Model-generated Summary:

[{'summary_text': "John is thinking about getting a PlayStation 5, but
he's
not sure if it's worth it as he doesn't have enough time to play it."}]
```

For this dialogue, I have decided to include some abbreviations such as *idk* — for *I don't know* — and *r u* — for *are you* — to observe how the model would interpret them.

We can see that the model has been able to successfully capture the essence of the dialogue and identify the main subject, which is John's uncertainty to buy a PlayStation 5 given the fact that he has so little time to play it.

```
Original Dialogue:

Camilla: Who do you think is going to win the competition?
Michelle: I believe Jonathan should win but I'm sure Mike is cheating!
Camilla: Why do you say that? Can you prove Mike is really cheating?
Michelle: I can't! But I just know!
Camilla: You shouldn't accuse him of cheating if you don't have any
evidence to support it.

Model-generated Summary:
[{'summary_text': 'Jonathan should win the competition, but Michelle
thinks
Mike is cheating.'}]
```

Once more the model captures the main theme of the conversation, which is Michelle's belief that Jonathan should win the competition, but that Mike may be cheating. Some further improvements could be made, though, such as including the information that Michelle cannot really show any evidence to support her belief that Mike is cheating.

Conclusion and Deployment

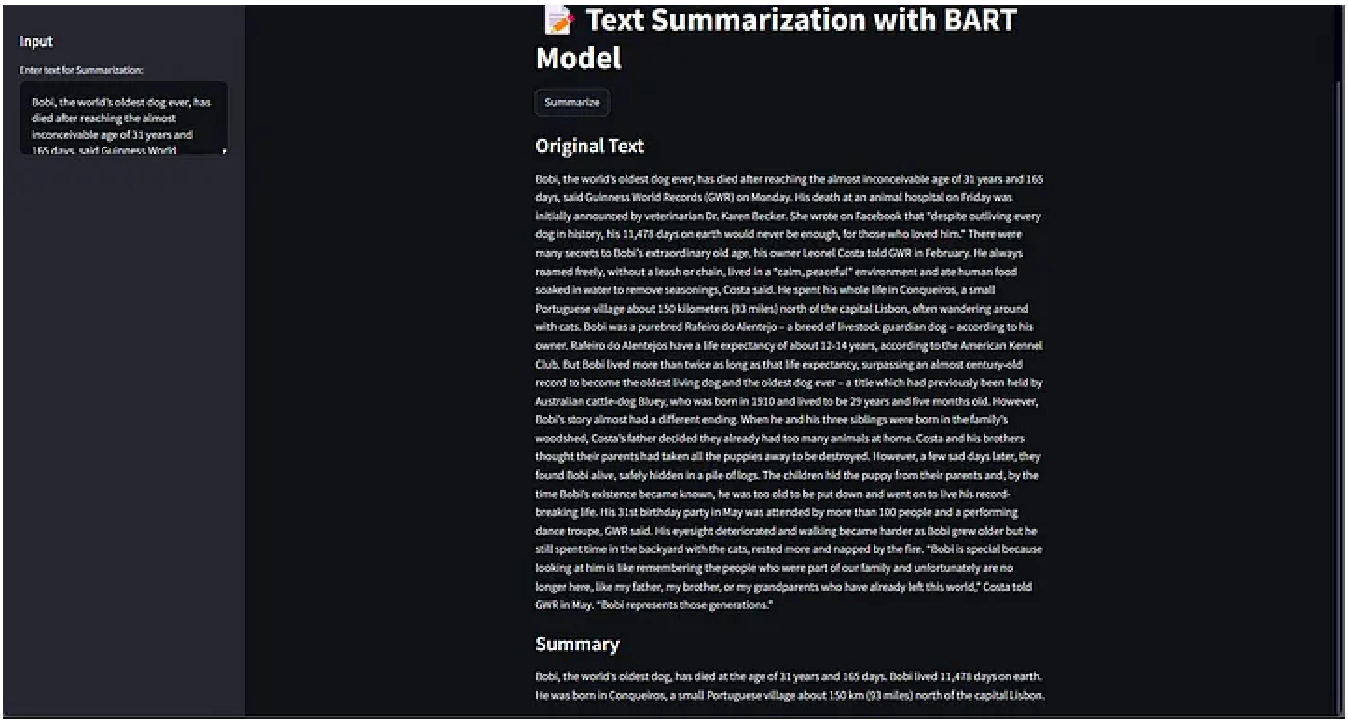
In this notebook, we have explored how we can use ***Large Language Models*** for several tasks involving Natural Language Processing, more specifically, Text Summarization tasks.

We delved into how Hugging Face's Transformers, Evaluate, and Datasets can be used to leverage frameworks such as PyTorch to fine-tune pre-trained models with a large number of parameters. This type of technique is usually referred to as **transfer learning**, which allows Data Scientists and Machine Learning Engineers to exploit the knowledge gained from previous tasks to improve generalization on a new task.

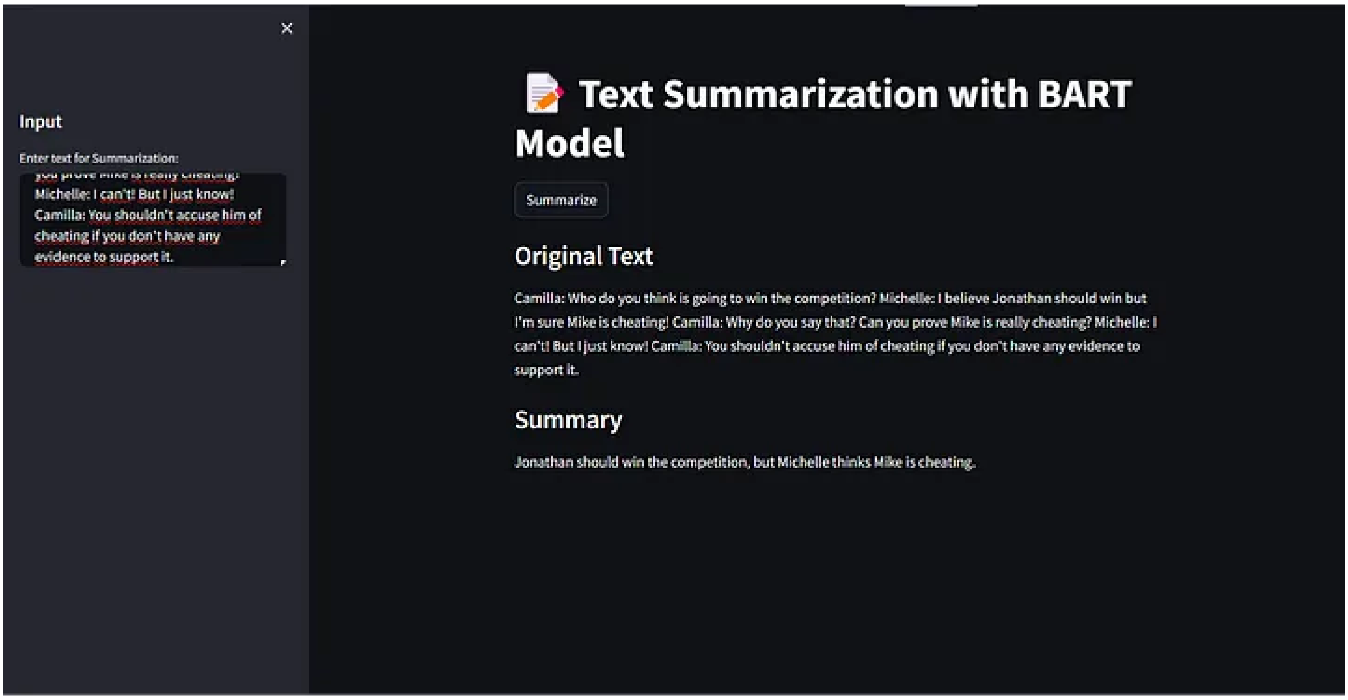
We used a BART model that has been already trained to perform summarization on news articles, and fine-tuned it to perform summarizations of dialogues with the **SamSum** dataset.

Thanks to Hugging Face's Models and Spaces, I have uploaded this model online, and it is free for anyone to use on their own summarization tasks or further fine-tune it on other tasks. I highly suggest you visit the luisotorres/bart-finetuned-samsum for more information on how to use this model.

I have also built a **web app** where you can use the model for the summarization of dialogues and news articles. Below, you can see some images of the web app, which is also available for free on Bart Text Summarization.



Example of Summarization of News Article



Example of Summarization of Dialogue

I hope that this notebook serves as a good introduction for those interested in the use of LLMs for Natural Language Processing tasks, as well as for those who already work with them and are in search of refining their knowledge on the subject.
This notebook took quite a while to be made and I highly appreciate your feedback on this work. Feel free to leave your comments, suggestions, and upvotes if you liked the content presented here.

Thank you very much!

Luis Fernando Torres

Let's connect! 🔗

LinkedIn • Kaggle • HuggingFace

Large Language Models

Deep Learning

Artificial Intelligence

Data Science

Machine Learning

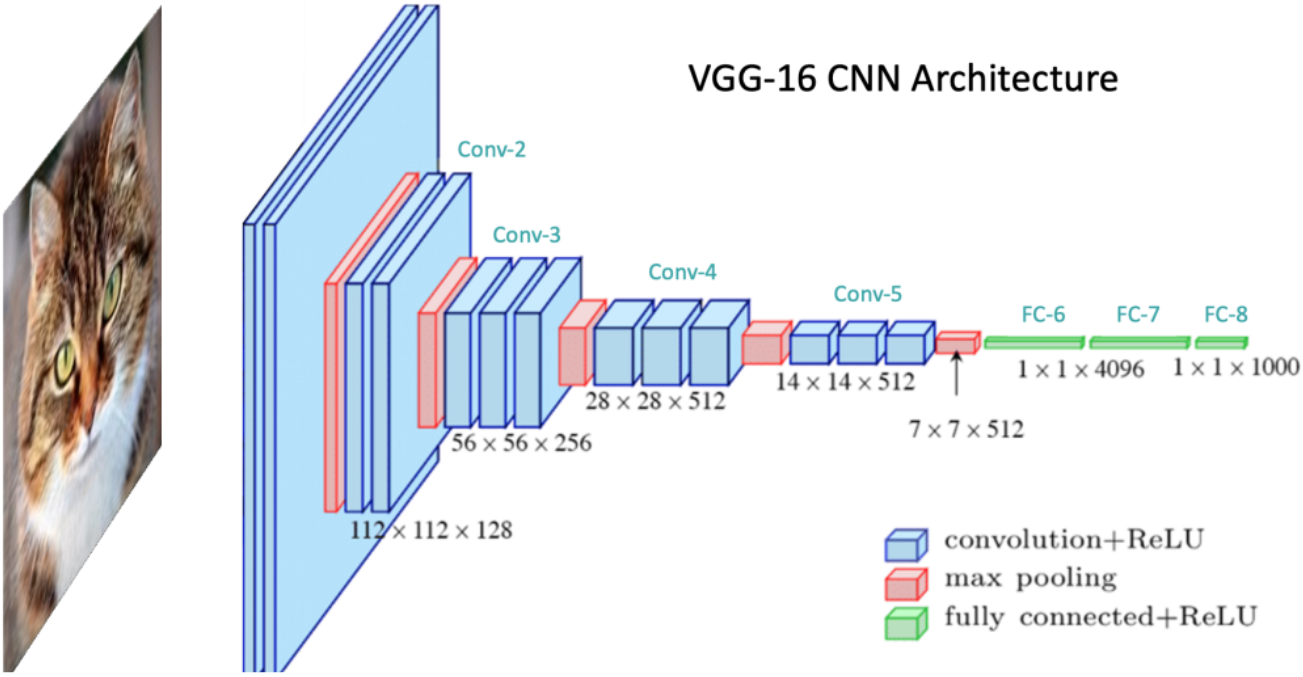
Follow

Written by Luís Fernando Torres

997 Followers

Data Scientist | Machine Learning Engineer | Commodities Trader & Investor | https://luuisotorres.github.io/

More from Luís Fernando Torres



Luís Fernando Torres in LatinXinAI

Convolutional Neural Network From Scratch

The most effective way of working with image data

21 min read · Oct 16

280 1

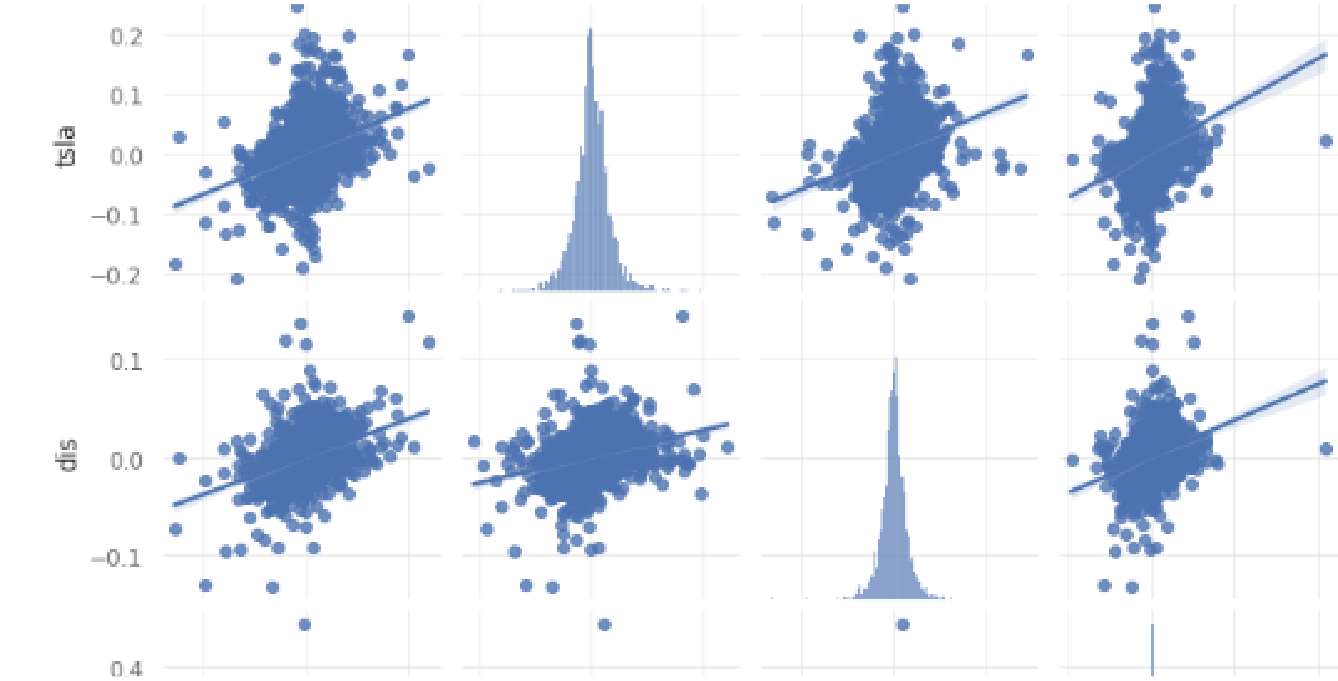
Luís Fernando Torres in LatinXinAI

How I Deployed a Machine Learning Model for the First Time

Going beyond Jupyter Notebooks 🚀

13 min read · Jul 18

848 8



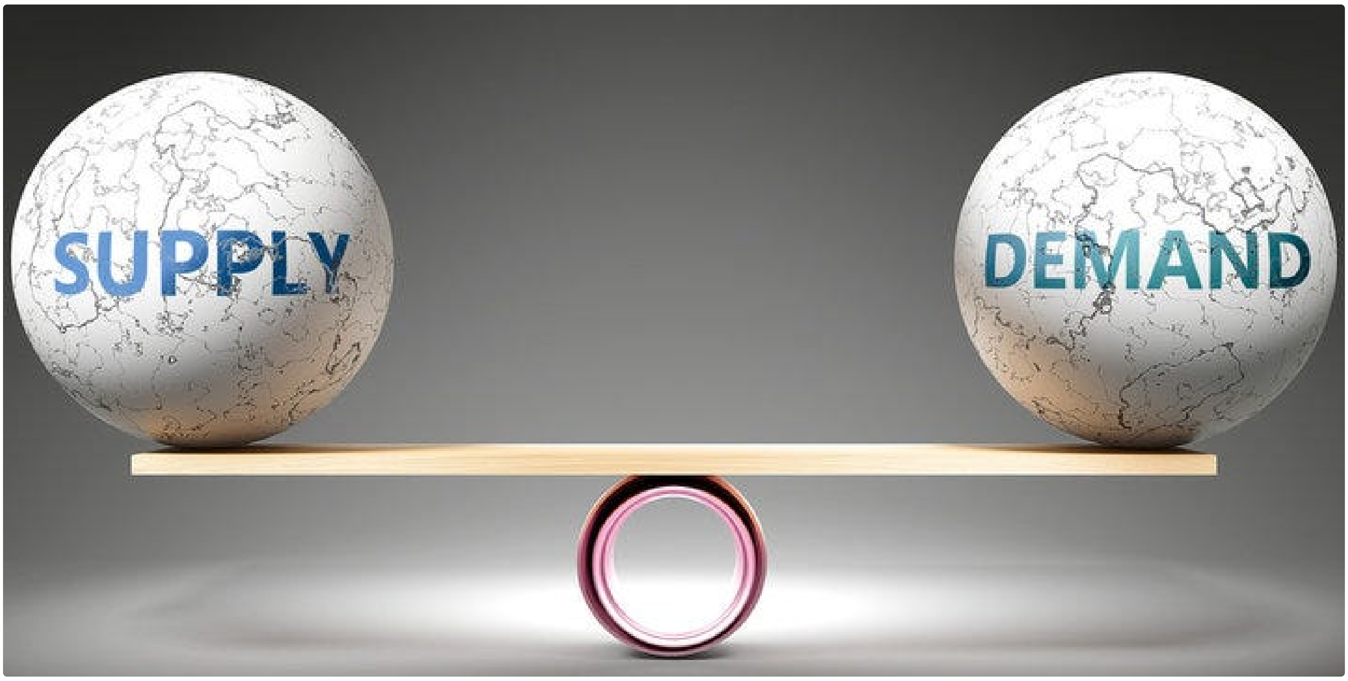
Luís Fernando Torres in InsiderFinance Wire

Introduction to Quant Investing with Python

Introduction

15 min read · Mar 30

1K 11



Luís Fernando Torres in InsiderFinance Wire

Volatility-Based Supply & Demand Levels Forecasting

Introduction

9 min read · Sep 21

150

See all from Luís Fernando Torres

Recommended from Medium