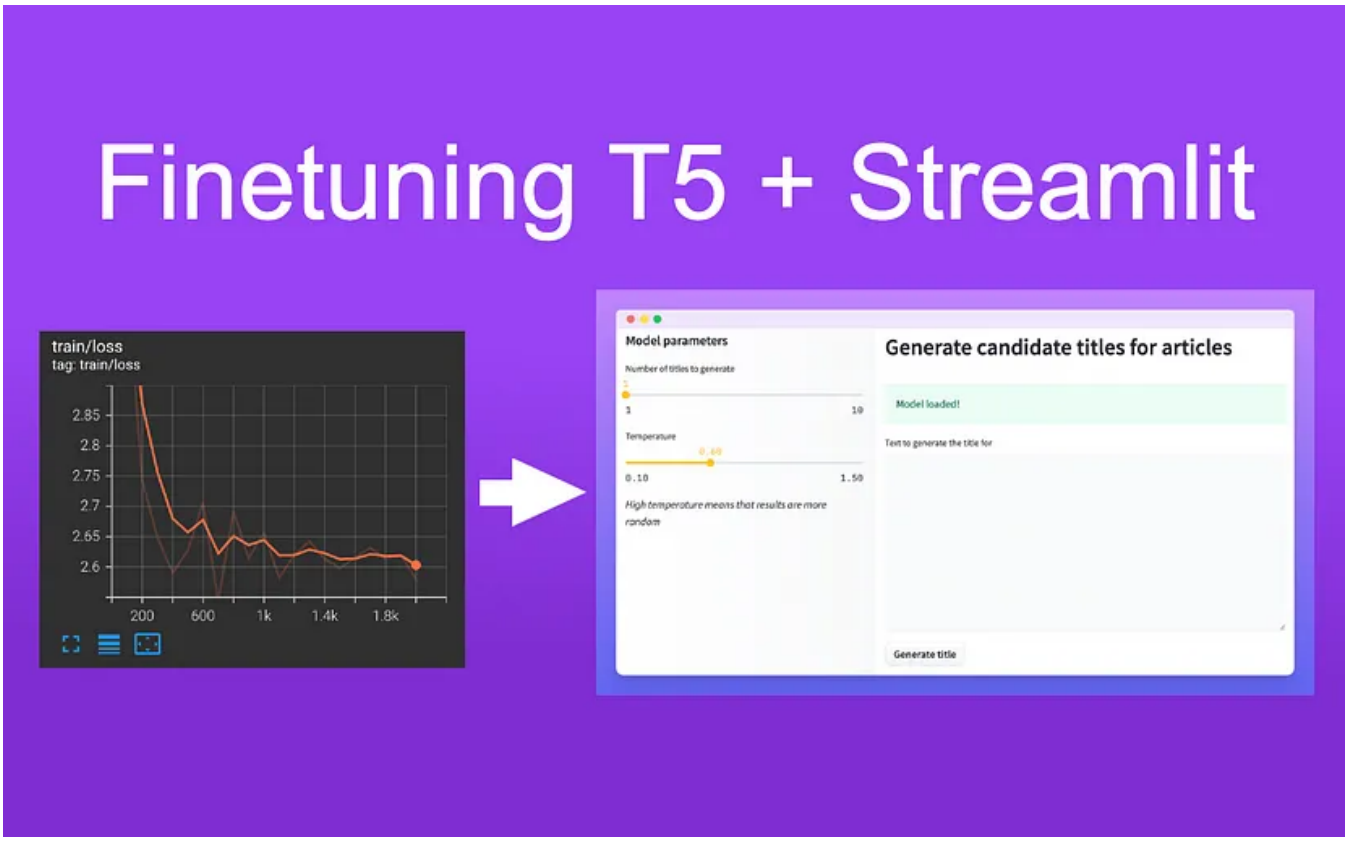
Published in NLPlanet

Fabio Chiusano Follow
May 17, 2022 · 15 min read · Listen

A Full Guide to Finetuning T5 for Text2Text and Building a Demo with Streamlit

All you need to know to build a full demo: Hugging Face Hub, Tensorboard, Streamlit, and Hugging Face Spaces



Hello fellow NLP enthusiasts! I'm back at writing articles after a small break, can't wait to dig into new NLP topics! In this article, we see a complete example of fine-tuning of T5 for generati 294 | 3 *s for articles. The model is fine-*

> *tuned entirely on Colab, we visualize its training with TensorBoard, upload the model on the Hugging Face Hub for everyone to use, and create a small demo with Streamlit that we upload on Hugging Face Spaces for everyone to try it out. Enjoy!* 😄

I've been wanting to experiment with Streamlit and Hugging Face Spaces for a while now. In case you didn't know them:

- Streamlit is a Python library that allows data scientists to easily create small interactive demos, so that other people can test their machine learning models or see their data analyses. Check out the Streamlit Gallery to learn what can be done with the library. Moreover, you can deploy your Streamlit application to the Streamlit Cloud, which makes it easy to share your applications with other people and you get free CPUs to run your applications. I suggest having a look at Gradio as well, which is another popular Python library with similar goals.
- Hugging Face Spaces is a service where you can deploy your Streamlit or Gradio applications so that you can easily share them. It provides free CPUs and it's similar to the Streamlit Cloud. However, when you deploy an application on Hugging Face Spaces that uses a machine learning model uploaded on the Hugging Face Hub, everyone can see that they are linked (i.e., that there is an interactive demo for that model).

To test them out, I decided to fine-tune a pre-trained model on a simple but not-so-popular task, which is the generation of candidate titles for articles starting from their textual content.

This is what we are going to do in this article:

1. Find a suitable dataset containing articles textual contents and titles.
2. Choose a suitable metric for our task.
3. Fine-tune a pre-trained model for title generation on Colab, monitoring the chosen metric on the validation set using TensorBoard, and saving the

model's checkpoints on Google Drive (so that we can resume training in case Colab shuts down the connection).

4. Upload the model on Hugging Face Hub for everyone to use.
5. Build an interactive demo with Streamlit and deploy it to Hugging Face Spaces.

Let's start!

Choosing a dataset for the Title Generation task

I usually look for datasets on Google Dataset Search or Kaggle. There are already available datasets with many newspaper articles, but for this tutorial I wanted to focus on the kind of articles that we usually find on Medium: less news, more guides.

Since I couldn't find a Medium articles dataset containing both articles textual contents and titles, I created one by myself by scraping the website and parsing web pages using the newspaper Python library. I then uploaded it to Kaggle with the name 190k+ Medium Articles dataset so that everyone can use it, along with a public notebook with simple data exploration.

Each row in the data is a different article published on Medium. For each article, you have the following features:

- *title [string]*: The title of the article.
- *text [string]*: The text content of the article.
- *url [string]*: The URL associated with the article.
- *authors [list of strings]*: The article authors.
- *timestamp [string]*: The publication datetime of the article.
- *tags [list of strings]*: List of tags associated with the article.

Choosing a metric for the Title Generation task

The task of generating titles starting from the textual content of an article is a text2text generation task: we have a text in input and we want to generate some text as output.

Popular text2text generation tasks are machine translation, commonly evaluated with the BLEU score and a focus on word precision, and text summarization, commonly evaluated with the ROUGE score and a focus on word recall.

I see title generation as closely related to text summarization as the title should make the reader understand what is the article about, with the added flavor that the title should also intrigue the reader and make him/her curious about the article. For this reason, I decided to evaluate my models with the ROUGE score.

Model training

Let's jump now into code with Colab! You can find all the code in this public Colab notebook as well. This is what we'll do:

1. Connect Google Drive to Colab to have persistent storage across Colab sessions.
2. Download the Medium dataset from Kaggle.
3. Load the dataset.
4. Split the dataset into train, validation, and test set.
5. Preprocess the dataset for T5.
6. Preparing the Hugging Face trainer.
7. Start TensorBoard.
8. Fine-tune T5.
9. Try the model.

10. Evaluate the model on the test set.

First, we install some libraries:

```
!pip install datasets transformers rouge_score nltk
```

title_prediction_2.sh hosted with ❤️ by GitHub view raw

We use the Hugging Face `transformers` library to download pre-trained models and fine-tune them, the Hugging Face `datasets` library to load our dataset and preprocess it, the `rouge-score` library to compute ROUGE scores, and the `nltk` library which contains common NLP preprocessing functions.

Connect Google Drive to Colab

Here is a Colab notebook that provides recipes for loading and saving data in Colab from external sources. We are interested in the "Mounting Google Drive locally" section: by executing the following code in our notebook, you'll be prompted to allow the notebook to read data from your Google Drive.

```
from google.colab import drive
drive.mount('/content/drive')
```

title_prediction_1.py hosted with ❤️ by GitHub view raw

You can then find your Google Drive data in the `drive/MyDrive` directory. Later in this article, we'll save our model checkpoints inside a purposely created `drive/MyDrive/Models` directory.

Download the Medium dataset from Kaggle

We'll download our dataset using the `kaggle` command-line utility, which comes already installed on Google Colab. To use the `kaggle` command, we need to connect it to our Kaggle account through an API key. Here are the instructions to create the API key, it's very easy. Once done, you should have a `kaggle.json` file.

Next, you need to upload your `kaggle.json` file to your Colab instance. If you don't know how to do it, check the "Uploading files from your local file system" section of this notebook.

Lastly, we need to instruct the `kaggle` command-line utility to use your `kaggle.json` file. You can do this by executing the command `kaggle` , which will create the hidden directory `.kaggle` (along with throwing an error because it can't find a `kaggle.json` file inside the `.kaggle` directory). Then, copy the file into the directory with `cp kaggle.json ~/.kaggle/kaggle.json` .

We can now download the dataset! First, we must retrieve the dataset name, which is whatever comes after `kaggle.com/datasets/` in the dataset URL. Our dataset URL is `kaggle.com/datasets/fabiochiusano/medium-articles` , therefore the dataset name is `fabiochiusano/medium-articles` . Execute the command `kaggle datasets download -d fabiochiusano/medium-articles` , and you should find the file `medium-articles.zip` in your current directory.

Load the dataset

Let's import the necessary libraries.

```
1  import transformers
2  from datasets import load_dataset, load_metric
```

title_prediction_3.py hosted with ❤️ by GitHub view raw

We load the dataset using the `load_dataset` function from the `datasets` package.

```python
medium_datasets = load_dataset("csv", data_files="medium-articles.zip")
# DatasetDict({
#     train: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
#         num_rows: 192368
#     })
# })
```

title_prediction_4.py hosted with ❤️ by GitHub view raw

Split the dataset into train, validation, and test set

As a common practice, we split the dataset into:

- *Training set*: Data used for training the model parameters.
- *Validation set*: Data used for hyperparameter tuning or early stopping to avoid overfitting.
- *Test set*: Data used for checking what performance we can expect on new data.

This can be done with the `train_test_split` methods of our dataset object, as follows.

```python
datasets_train_test = medium_datasets["train"].train_test_split(test_size=3000)
datasets_train_validation = datasets_train_test["train"].train_test_split(test_size=300

medium_datasets["train"] = datasets_train_validation["train"]
medium_datasets["validation"] = datasets_train_validation["test"]
medium_datasets["test"] = datasets_train_test["test"]

# DatasetDict({
#     train: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
#         num_rows: 186368
#     })
#     validation: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
#         num_rows: 3000
#     })
#     test: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
#         num_rows: 3000
#     })
# })
```

title_prediction_5.py hosted with ❤️ by GitHub view raw

The choice of the sizes of the three datasets usually depends on many factors, such as the size of the whole dataset, how much time you want to spend training or evaluating your model, and how precise you want your model

evaluations to be. Since we are training a quite big model on a free GPU from Colab, I decided to use small validation and test sets to speed things up. Despite this, we will see that the final model will be able to produce good titles.

```python
medium_datasets["train"] = medium_datasets["train"].shuffle().select(range(100000))
medium_datasets["validation"] = medium_datasets["validation"].shuffle().select(range(10
medium_datasets["test"] = medium_datasets["test"].shuffle().select(range(1000))

# DatasetDict({
#     train: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
#         num_rows: 100000
#     })
#     validation: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
#         num_rows: 1000
#     })
#     test: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
#         num_rows: 1000
#     })
# })
```

title_prediction_6.py hosted with ❤️ by GitHub view raw

Preprocess the dataset for T5

Hugging Face provides us with a complete notebook example of how to fine-tune T5 for text summarization. As for every transformer model, we need first to tokenize the textual training data: the article content and the title.

Let's instantiate the tokenizer of the T5-base model.

```
import nltk
nltk.download('punkt')
import string
from transformers import AutoTokenizer

model_checkpoint = "t5-base"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
```

title_prediction_7.py hosted with ❤️ by GitHub view raw

Before applying the tokenizer to the data, let's filter out some bad samples (i.e. articles whose title is long less than 20 characters and whose text content is long less than 500 characters).

```
medium_datasets_cleaned = medium_datasets.filter(
    lambda example: (len(example['text']) >= 500) and
    (len(example['title']) >= 20)
)
```

title_prediction_9.py hosted with ❤️ by GitHub view raw

Then, we define the `preprocess_data` function that takes a batch of samples as inputs and outputs a dictionary of new features to add to the samples. The `preprocess_data` function does the following:

1. Extract the “text” feature from each sample (i.e. the article text content), fix the newlines in the article, and remove the lines without ending punctuation (i.e. the subtitles).

2. Prepend the text *“summarize: “* to each article text, which is needed for fine-tuning T5 on the summarization task.

3. Apply the T5 tokenizer to the article text, creating the `model_inputs` object. This object is a dictionary containing, for each article, an `input_ids` and an `attention_mask` arrays containing the token ids and the attention masks respectively.

4. Apply the T5 tokenizer to the article titles, creating the `labels` object. Also in this case, this object is a dictionary containing, for each article, an `input_ids` and an `attention_mask` arrays containing the token ids and the attention masks respectively. Note that this step is done inside the `tokenizer.as_target_tokenizer()` context manager: this is usually done because there are text2text tasks where inputs and labels must be tokenized with different tokenizers (e.g. when translating between two languages, where each language has its own tokenizer). As far as I know, for text summarization the labels are tokenized with the same tokenizer as the inputs, thus the context manager is optional.

5. Return a dictionary containing the token ids and attention masks of the inputs, and the token ids of the labels.

```
prefix = "summarize: "
max_input_length = 512
max_target_length = 64

def clean_text(text):
  sentences = nltk.sent_tokenize(text.strip())
  sentences_cleaned = [s for sent in sentences for s in sent.split("\n")]
  sentences_cleaned_no_titles = [sent for sent in sentences_cleaned
                                 if len(sent) > 0 and
                                 sent[-1] in string.punctuation]
  text_cleaned = "\n".join(sentences_cleaned_no_titles)
  return text_cleaned

def preprocess_data(examples):
  texts_cleaned = [clean_text(text) for text in examples["text"]]
  inputs = [prefix + text for text in texts_cleaned]
  model_inputs = tokenizer(inputs, max_length=max_input_length, truncation=True)

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

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

> *Note that we are truncating the inputs at 512 tokens. While T5 can manage longer inputs, the memory requirements grow quadratically with the size of the inputs, and that was the maximum size that I could use in my Colab session. 512 tokens correspond to about 682 English words, which is more or less what an average person reads in two minutes. The majority of Medium articles have between four and seven minutes of reading time, therefore we are currently throwing away useful information for our task. Despite this, many articles state what they are about in their first paragraphs, therefore good titles can be generated on most occasions.*

The `preprocess_data` function can be applied to all the datasets with the `map` method.

```python
tokenized_datasets = medium_datasets_cleaned.map(preprocess_function,
                                                 batched=True)

# DatasetDict({
#     train: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags',
#                    'input_ids', 'attention_mask', 'labels'],
#         num_rows: 85602
#     })
#     validation: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags',
#                    'input_ids', 'attention_mask', 'labels'],
#         num_rows: 860
#     })
#     test: Dataset({
#         features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags',
#                    'input_ids', 'attention_mask', 'labels'],
#         num_rows: 844
#     })
# })
```


Preparing the Hugging Face trainer

We can now fine-tune T5 with our preprocessed data! Let’s import some necessary classes to train text2text models.

```
1  from transformers import AutoModelForSeq2SeqLM, DataCollatorForSeq2Seq,
2    Seq2SeqTrainingArguments, Seq2SeqTrainer
```

title_prediction_11.py hosted with ❤️ by GitHub view raw

Next, we need to create a `Seq2SeqTrainingArguments` object containing, as the name implies, several training parameters that will define how the model is trained. Refer to the `Trainer` documentation to know about the meaning of each one of these parameters.

```python
batch_size = 8
model_name = "t5-base-medium-title-generation"
model_dir = f"drive/MyDrive/Models/{model_name}"

args = Seq2SeqTrainingArguments(
    model_dir,
    evaluation_strategy="steps",
    eval_steps=100,
    logging_strategy="steps",
    logging_steps=100,
    save_strategy="steps",
    save_steps=200,
    learning_rate=4e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    weight_decay=0.01,
    save_total_limit=3,
    num_train_epochs=1,
    predict_with_generate=True,
    fp16=True,
    load_best_model_at_end=True,
    metric_for_best_model="rouge1",
    report_to="tensorboard"
)
```

Here is the explanation of some unusual parameters passed to the `Seq2SeqTrainingArguments` object:

- `predict_with_generate` : Must be set to `True` to calculate generative metrics such as ROUGE and BLEU.
- `fp16` : Whether to use fp16 16-bit (mixed) precision training instead of 32-bit training. Makes training faster.
- `report_to` : List of integrations to write logs to.

Note that the output directory is inside Google Drive and that we are specifying `rouge1` as the metric which defines the best model.

Next, we instantiate a `DataCollatorForSeq2Seq` object using the tokenizer. Data collators are objects that form a batch by using a list of dataset elements as input and, in some cases, applying some processing. In this case, all the inputs and labels in the same batch will be padded to their respective maximum length in the batch. Padding of the inputs is done with the usual *[PAD]* token, whereas the padding of the labels is done with a token with id *-100*, which is a special token automatically ignored by PyTorch loss functions.

Next, we download the ROUGE code using the `load_metric` function from the `datasets` library, thus instantiating a metric object. This object can then be used to compute its metrics using predictions and reference labels.

The `metric` object must then be called inside a `compute_metrics` function which takes a tuple of predictions and reference labels as input, and outputs a dictionary of metrics computed over the inputs. Specifically, the `compute_metrics` function does the following:

1. Decode the predictions (i.e. from token ids to words).
2. Decode the labels after substituting the *-100* token id with the *[PAD]* token id.
3. Compute ROUGE scores using the decoded predictions and labels, and select only a subset of these metrics.

4. Compute a new metric, which is the average length of the predictions.
5. Return a dictionary whose keys are the names of the metrics and the values are the metric values.

We are almost done! We must now create a `Seq2SeqTrainer` passing all the objects that we have just defined: the training arguments, the training and evaluation data, the data collator, the tokenizer, the `compute_metrics` function, and a `model_init` function. The `model_init` function must return a fresh new instance of the pre-trained model to fine-tune, making sure that training always starts from the same model and not from a partially fine-tuned model from your notebook.

Start TensorBoard

Before starting the training, let's start TensorBoard inside our notebook.

After executing this code, the notebook should display TensorBoard right under the cell.

Fine-tune T5

Last, we start the fine-tuning with the `train` method.

Once started, the cell will output a progress bar indicating the number of batches processed. Moreover, at each evaluation cycle, we'll see also the values of our metrics on the validation set, which you can see in the next image.

Training and evaluation metrics logged during fine-tuning.

The number of each step represents the number of batches processed. For example, at step 100, the model has been trained on `100 * batch_size` samples, which in this case is `100 * 8 = 800`.

Both the training and evaluation losses gradually decrease up to step 2100. However, all the ROUGE metrics seem to have peaked around step 1500 and later decreased a little. Since the training already took around three hours, I decided to stop the training and test the results, but it's not impossible that the ROUGE scores could have risen again after some steps, thus making me miss a better model.

The average length of the generated titles has always remained stable at around 12 tokens. As each token with subword tokenizers is more or less long 0.75 English words, we can make a quick estimate that the average length of the generated articles is around 9 words (looks ok!).

Here are some charts extracted from TensorBoard.

Training loss during fine-tuning.

The training loss seems to stabilize around step 1700, slowly decreasing.

Average length of the generated titles and loss over the evaluation set during fine-tuning.

The evaluation loss is still decreasing at step 2000. The average length of the generated titles remains somewhat stable at 12 tokens during training.

ROUGE-1 and ROUGE-2 over the evaluation set during fine-tuning.

ROUGE-1 stabilizes at 33% at step 1800, whereas ROUGE-2 stabilizes at 17.5% at step 1500.

ROUGE-L and ROUGE-L-sum over the evaluation set during fine-tuning.

ROUGE-L and ROUGE-L-sum both stabilize at 30.7% at step 1500. While ROUGE-L computes the longest common subsequence (LCS) between the generated text and the reference text ignoring newlines, ROUGE-L-sum does the same considering newlines as sentence boundaries.

Try the model

We can now test our model! First, let's load it.

Let’s try it on the webpage “Add statefulness to apps” of the Streamlit documentation.

Our model predicts the title “Session State and Callbacks in Streamlit”. Looks good!

We test it on the Medium article “Banking on Bots” as a second example.

The model predicts the title “Conversational AI: The Future of Customer Service”, which is coherent with the article content and has proper capitalization as well. It looks like the model is working as expected.

Evaluate the model on the test set

Last, let’s compute the metrics on the test set with the new model to check that training has been done correctly and that we can expect good performance on new unseen data.

We need to:

1. Pad the articles texts to the same length.

2. Group the samples into batches (using a PyTorch dataloader).
3. Generate a title for each sample.
4. Compare the generated titles with the reference titles using the `compute_metrics` function defined earlier.

The metrics are better than the ones from the validation set. We probably need to increase the size of both the validation and test set to have less variable results, but let's consider these results satisfactory for now.

Uploading the model to Hugging Face Hub

Loading the model from Google Drive is tedious since we need to mount it to the Colab filesystem every time. Let's see how to upload our model on Hugging Face Hub.

When uploading your model to the Hugging Face Hub, it's good practice to upload it with each one of the three main deep learning frameworks managed by Hugging Face: PyTorch, TensorFlow, and JAX. Thus, we need to install JAX to convert our model, as it's not installed in Colab by default.

Then, we need to authenticate to our Hugging Face account, as the uploaded models will be linked to it.

Last, we upload our model in the three formats. Remember to upload the tokenizer as well!

Once completed, this is how we can load the model and the tokenizer from the hub. Note that the name of the model is preceded by the name of the account linked to it.

Remember to update the model card of your uploaded model. This step consists in updating the README inside your model repository. Have a look at the final model card of our model!

Build an interactive demo with Streamlit and deploy it to Hugging Face Spaces

Let's see again the line of code that we use to generate a candidate title for an article.

As you can see, we are using several parameters of the `generate` method to tune the results. A common parameter used when generating text is `temperature`, which controls the tradeoff between obtaining deterministically the most probable results and obtaining plausible different (but somewhat less probable) results every time we run the function. Indeed, we may want to generate 10 candidate titles instead of one.

Moreover, our model considers only the first 512 tokens of the article text when generating candidate titles. What if we want to generate multiple titles, where each title is generated taking into account 512 tokens from distinct parts of the article? We would need to add custom prediction logic to the model, which is not easily testable directly from the Hugging Face Hub.

However, there's a way to allow users to experiment with these settings: we could build a small demo with this custom logic using Streamlit!

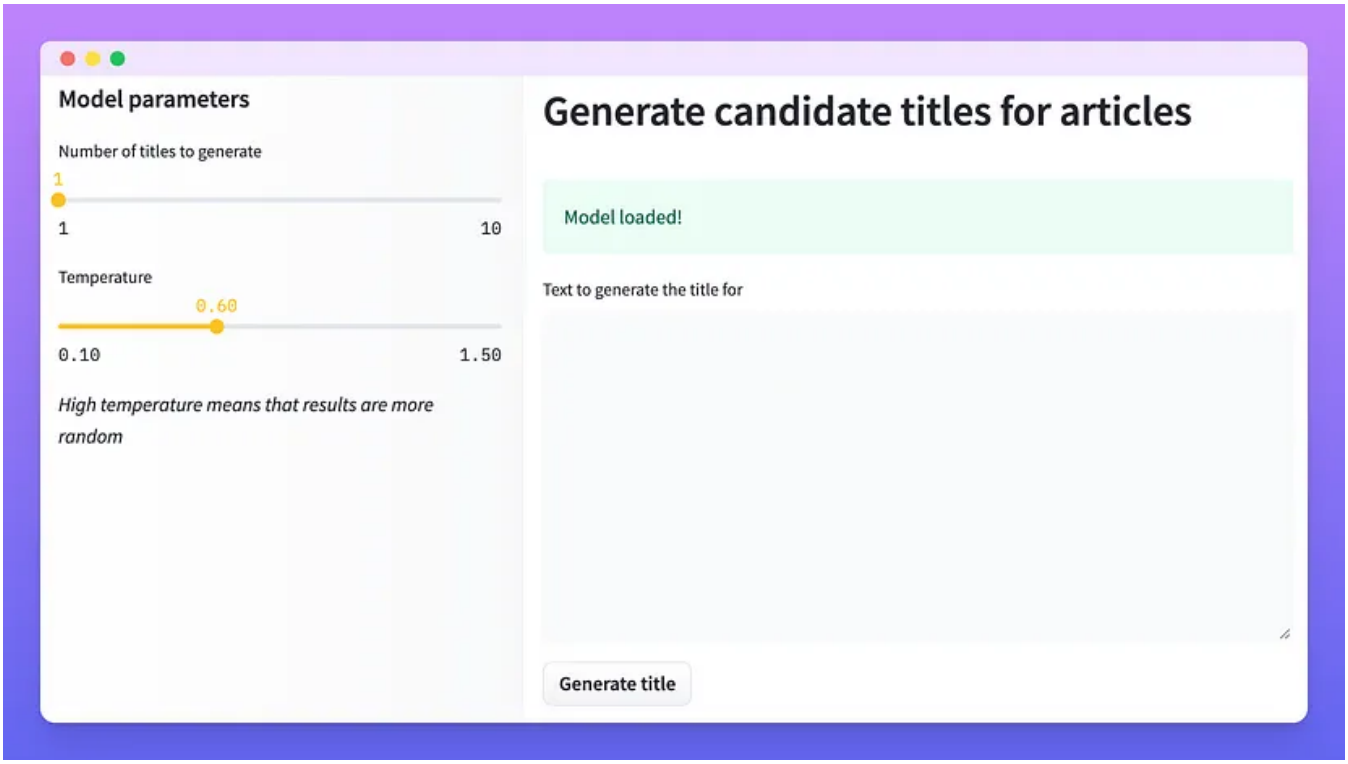
I'm not going into code details with Streamlit, but rest assured that it's very easy to use and quick to learn. I suggest reading the Streamlit documentation, it takes a couple of hours.

Have a look at this repo to see the complete code of our Streamlit application. There are two main components:

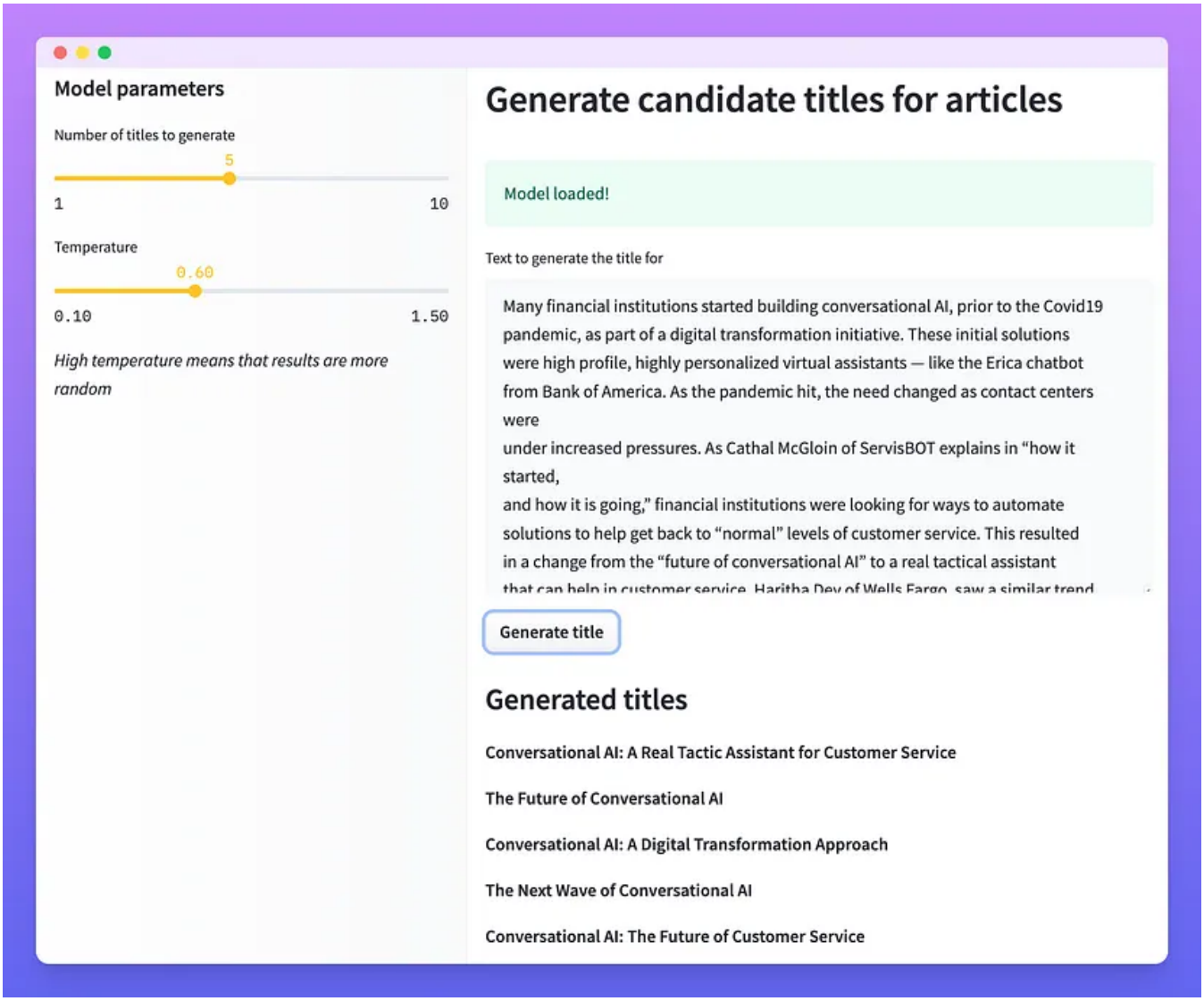
- The `app.py` file: Contains the Streamlit code of the app. It's where we load our model from the Hugging Face Hub, create some interactive components, and write some logic to connect the two.
- The `requirements.py` file: Contains all the Python libraries used in the `app.py` file (such as the `transformers` library). It's not necessary to add the Streamlit library to this file.

Once we have tested locally our Streamlit application and everything works as expected, we need to upload it to a new Hugging Face Space. This step is just a matter of copying the files of your Streamlit applications to a new repo created on your Hugging Face account. Refer to this article to learn how to host your Streamlit projects in Hugging Face Spaces.

This is the resulting Hugging Face Space which uses our title generation model in the background and allows generate multiple titles (taking into account different spans of the article text) and change the `temperature` parameter.



Let's try generating *5* candidate titles for the "Banking on Bots" Medium article, using a `temperature` of *0.6*.



The generated titles are:

- Conversational AI: A Real Tactic Assistant for Customer Service
- The Future of Conversational AI
- Conversational AI: A Digital Transformation Approach
- The Next Wave of Conversational AI
- Conversational AI: The Future of Customer Service

Building a small demo with Streamlit is very easy and, in conjunction with Hugging Face Spaces or Streamlit Cloud, allows other people to test your work without needing the knowledge of how to run a Jupyter notebook.

Conclusion and next steps

In this article, we chose a suitable dataset and metric for our title generation task, and we wrote some code with the Hugging Face library to fine-tune a pre-trained T5 model for our task. We then uploaded our new model to the Hugging Face Hub so that everybody can use it, and we made a demo application with Streamlit which is now hosted as a Hugging Face Space.

I believe Hugging Face Space is an effortless way to build a portfolio of NLP applications where people or organizations can show off their machine learning skills.

Possible next steps are:

- Fine-tune a Longformer Encoder Decoder (LED) model instead of T5, as it is able to use a longer context as input. Keep in mind that the training will be slower though.
- Fine-tune a BART model and compare the results against the fine-tuned T5 model.

Thank you for reading! If you are interested in learning more about NLP, remember to follow NLPlanet on Medium, LinkedIn, Twitter, and join our new Discord server!

Data Science Artificial Intelligence NLP Machine Learning Hugging Face