TDS Archive

Choosing and Implementing Hugging Face Models

Pulling pre-trained models out of the box for your use case

Stephanie Kirmer Follow 8 min read · Nov 1, 2024

Photo by Erda Estremera on Unsplash

I've been having a lot of fun in my daily work recently experimenting with models from the Hugging Face catalog, and I thought this might be a good time to share what I've learned and give readers some tips for how to apply these models with a minimum of stress.

My specific task recently has involved looking at blobs of unstructured text

data (think memos, emails, free text comment fields, etc) and classifying them according to categories that are relevant to a business use case. There are a ton of ways you can do this, and I've been exploring as many as I can feasibly do, including simple stuff like pattern matching and lexicon search, but also expanding to using pre-built neural network models for a number of different functionalities, and I've been moderately pleased with the results.

I think the best strategy is to incorporate multiple techniques, in some form of ensembling, to get the best of the options. I don't trust these models necessarily to get things right often enough (and definitely not consistently enough) to use them solo, but when combined with more basic techniques they can add to the signal.

Choosing the use case

For me, as I've mentioned, the task is just to take blobs of text, usually written by a human, with no consistent format or schema, and try to figure out what categories apply to that text. I've taken a few different approaches, outside of the analysis methods mentioned earlier, to do that, and these range from very low effort to somewhat more work on my part. These are three of the strategies that I've tested so far.

- Ask the model to choose the category (zero-shot classification — I'll use this as an example later on in this article)
- Use a named entity recognition model to find key objects referenced in the text, and make classification based on that
- Ask the model to summarize the text, then apply other techniques to make classification based on the summary

Finding the models

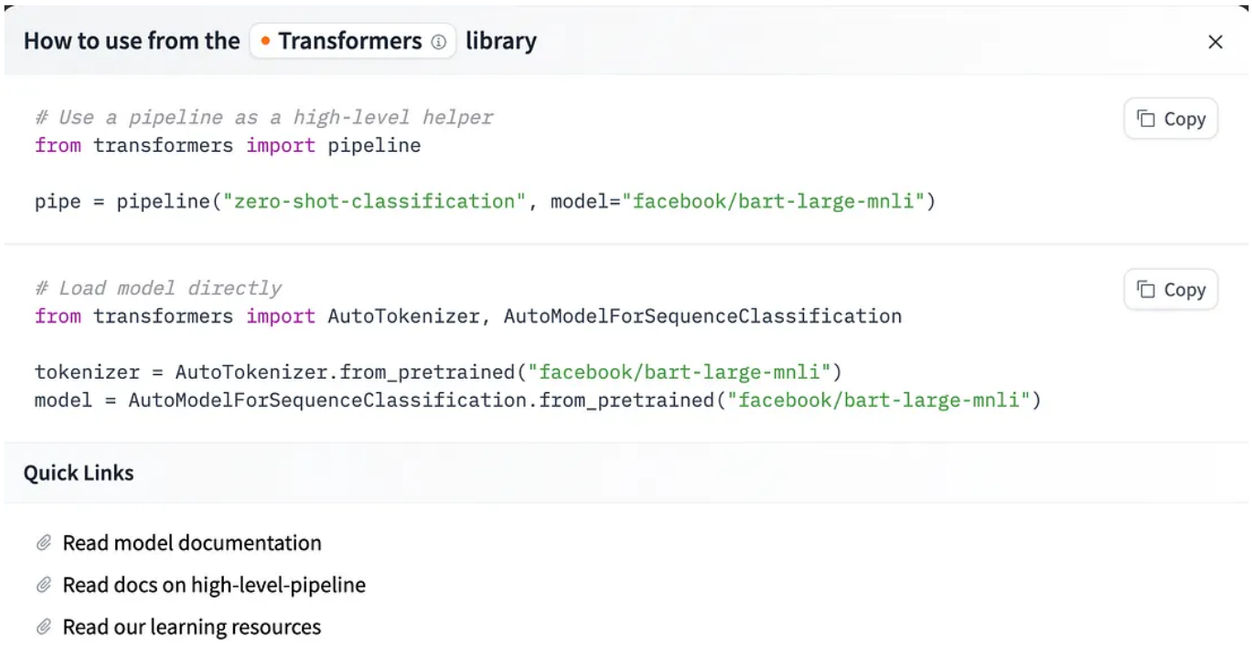
This is some of the most fun — looking through the Hugging Face catalog for models! At https://huggingface.co/models you can see a gigantic assortment of the models available, which have been added to the catalog by users. I have a few tips and pieces of advice for how to select wisely.

- Look at the download and like numbers, and don't choose something that has not been tried and tested by a decent number of other users. You can also check the Community tab on each model page to see if users are discussing challenges or reporting bugs.

- Investigate who uploaded the model, if possible, and determine if you find them trustworthy. This person who trained or tuned the model may or may not know what they're doing, and the quality of your results will depend on them!
- Read the documentation closely, and skip models with little or no documentation. You'll struggle to use them effectively anyway.
- Use the filters on the side of the page to narrow down to models suited to your task. The volume of choices can be overwhelming, but they are well categorized to help you find what you need.
- Most model cards offer a quick test you can run to see the model's behavior, but keep in mind that this is just one example and it's probably one that was chosen because the model's good at that and finds this case pretty easy.

Incorporating into your code

Once you've found a model you'd like to try, it's easy to get going- click the "Use this Model" button on the top right of the Model Card page, and you'll see the choices for how to implement. If you choose the Transformers option, you'll get some instructions that look like this.



Screenshot taken by author

If a model you've selected is not supported by the Transformers library, there may be other techniques listed, like TF-Keras, scikit-learn, or more, but all should show instructions and sample code for easy use when you click that button.

In my experiments, all the models were supported by Transformers, so I had a mostly easy time getting them running, just by following these steps. If you find that you have questions, you can also look at the deeper documentation and see full API details for the Transformers library and the different classes it offers. I've definitely spent some time looking at these docs for specific classes when optimizing, but to get the basics up and running you shouldn't really need to.

Preparing inference data

Ok, so you've picked out a model that you want to try. Do you already have data? If not, I have been using several publicly available datasets for this experimentation, mainly from Kaggle, and you can find lots of useful datasets there as well. In addition, Hugging Face also has a dataset catalog you can check out, but in my experience it's not as easy to search or to understand the data contents over there (just not as much documentation).

Once you pick a dataset of unstructured text data, loading it to use in these models isn't that difficult. Load your model and your tokenizer (from the docs provided on Hugging Face as noted above) and pass all this to the `pipeline` function from the transformers library. You'll loop over your blobs of text in a list or pandas Series and pass them to the model function. This is essentially the same for whatever kind of task you're doing, although for zero-shot classification you also need to provide a candidate label or list of labels, as I'll show below.

Code Example

So, let's take a closer look at zero-shot classification. As I've noted above, this involves using a pretrained model to classify a text according to categories that it hasn't been specifically trained on, in the hopes that it can use its learned semantic embeddings to measure similarities between the text and the label terms.

```
from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenizer
from transformers import pipeline

nli_model = AutoModelForSequenceClassification.from_pretrained("facebook/bart-la
tokenizer = AutoTokenizer.from_pretrained("facebook/bart-large-mnli")
classifier = pipeline("zero-shot-classification", device="cpu", model=nli_model,
```

```
label_list = ['News', 'Science', 'Art']

all_results = []
for text in list_of_texts:
    prob = self.classifier(text, label_list, multi_label=True, use_fast=True)
    results_dict = {x: y for x, y in zip(prob["labels"], prob["scores"])}
    all_results.append(results_dict)
```

This will return you a list of dicts, and each of those dicts will contain keys for the possible labels, and the values are the probability of each label. You don't have to use the pipeline as I've done here, but it makes multi-label zero shot a lot easier than manually writing that code, and it returns results that are easy to interpret and work with.

If you prefer to not use the pipeline, you can do something like this instead, but you'll have to run it once for each label. Notice how the processing of the logits resulting from the model run needs to be specified so that you get human-interpretable output. Also, you still need to load the tokenizer and the model as described above.

```
def run_zero_shot_classifier(text, label):
    hypothesis = f"This example is related to {label}."

    x = tokenizer.encode(
        text,
        hypothesis,
        return_tensors="pt",
        truncation_strategy="only_first"
    )

    logits = nli_model(x.to("cpu"))[0]

    entail_contradiction_logits = logits[:, [0, 2]]
    probs = entail_contradiction_logits.softmax(dim=1)
    prob_label_is_true = probs[:, 1]

    return prob_label_is_true.item()

label_list = ['News', 'Science', 'Art']
all_results = []
for text in list_of_texts:
    for label in label_list:
        result = run_zero_shot_classifier(text, label)
            all_results.append(result)
```

To tune, or not?

You probably have noticed that I haven't talked about fine tuning the models

myself for this project — that's true. I may do this in future, but I'm limited by the fact that I have minimal labeled training data to work with at this time. I can use semisupervised techniques or bootstrap a labeled training set, but this whole experiment has been to see how far I can get with straight off-the-shelf models. I do have a few small labeled data samples, for use in testing the models' performance, but that's nowhere near the same volume of data I will need to tune the models.

If you do have good training data and would like to tune a base model, Hugging Face has some docs that can help. https://huggingface.co/docs/transformers/en/training

Computation and speed

Performance has been an interesting problem, as I've run all my experiments on my local laptop so far. Naturally, using these models from Hugging Face will be much more compute intensive and slower than the basic strategies like regex and lexicon search, but it provides signal that can't really be achieved any other way, so finding ways to optimize can be worthwhile. All these models are GPU enabled, and it's very easy to push them to be run on GPU. (If you want to try it on GPU quickly, review the code I've shown above, and where you see "cpu" substitute in "cuda" if you have a GPU available in your programming environment.) Keep in mind that using GPUs from cloud providers is not cheap, however, so prioritize accordingly and decide if more speed is worth the price.

Most of the time, using the GPU is much more important for training (keep it in mind if you choose to fine tune) but less vital for inference. I'm not digging in to more details about optimization here, but you'll want to consider parallelism as well if this is important to you- both data parallelism and actual training/compute parallelism.

Testing and understanding output

We've run the model! Results are here. I have a few closing tips for how to review the output and actually apply it to business questions.

- Don't trust the model output blindly, but run rigorous tests and evaluate performance. Just because a transformer model does well on a certain text blob, or is able to correctly match text to a certain label regularly, doesn't mean this is generalizable result. Use lots of different examples

and different kinds of text to prove the performance is going to be sufficient.

- If you feel confident in the model and want to use it in a production setting, track and log the model's behavior. This is just good practice for any model in production, but you should keep the results it has produced alongside the inputs you gave it, so you can continually check up on it and make sure the performance doesn't decline. This is more important for these kinds of deep learning models because we don't have as much interpretability of why and how the model is coming up with its inferences. It's dangerous to make too many assumptions about the inner workings of the model.

As I mentioned earlier, I like using these kinds of model output as part of a larger pool of techniques, combining them in ensemble strategies — that way I'm not only relying on one approach, but I do get the signal those inferences can provide.

I hope this overview is useful for those of you getting started with pre-trained models for text (or other mode) analysis — good luck!

Read more of my work at www.stephaniekirmer.com.

Further Reading

Models - Hugging Face

We're on a journey to advance and democratize artificial intelligence through open source and open science.

huggingface.co

Model Parallelism

We're on a journey to advance and democratize artificial intelligence through open source and open science.

huggingface.co