CHAPTER 9

Dealing with Few to No Labels

There is one question so deeply ingrained into every data scientist's mind that it's usually the first thing they ask at the start of a new project: is there any labeled data? More often than not, the answer is "no" or "a little bit," followed by an expectation from the client that your team's fancy machine learning models should still perform well. Since training models on very small datasets does not typically yield good results, one obvious solution is to annotate more data. However, this takes time and can be very expensive, especially if each annotation requires domain expertise to validate.

Fortunately, there are several methods that are well suited for dealing with few to no labels! You may already be familiar with some of them, such as *zero-shot* or *few-shot learning*, as witnessed by GPT-3's impressive ability to perform a diverse range of tasks with just a few dozen examples.

In general, the best-performing method will depend on the task, the amount of available data, and what fraction of that data is labeled. The decision tree shown in Figure 9-1 can help guide us through the process of picking the most appropriate method.

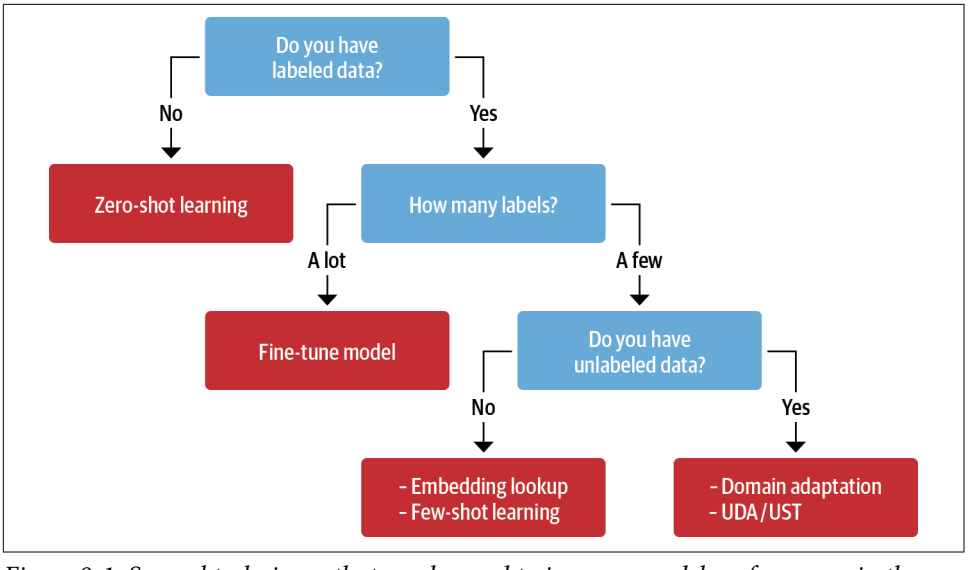


Figure 9-1. Several techniques that can be used to improve model performance in the absence of large amounts of labeled data

Let's walk through this decision tree step-by-step:

1. Do you have labeled data?
: Even a handful of labeled samples can make a difference with regard to which method works best. If you have no labeled data at all, you can start with the zero-shot learning approach, which often sets a strong baseline to work from.

2. How many labels?
: If labeled data is available, the deciding factor is how much. If you have a lot of training data available you can use the standard fine-tuning approach discussed in Chapter 2.

3. Do you have unlabeled data?
: If you only have a handful of labeled samples it can help immensely if you have access to large amounts of unlabeled data. If you have access to unlabeled data you can either use it to fine-tune the language model on the domain before training a classifier, or you can use more sophisticated methods such as unsupervised data augmentation (UDA) or uncertainty-aware self-training (UST).[1] If you don't have any unlabeled data available, you don't have the option of annotating more

1 Q. Xie et al., "Unsupervised Data Augmentation for Consistency Training", (2019); S. Mukherjee and A.H. Awadallah, "Uncertainty-Aware Self-Training for Few-Shot Text Classification", (2020).

data. In this case you can use few-shot learning or use the embeddings from a pretrained language model to perform lookups with a nearest neighbor search.

In this chapter we'll work our way through this decision tree by tackling a common problem facing many support teams that use issue trackers like Jira or GitHub to assist their users: tagging issues with metadata based on the issue's description. These tags might define the issue type, the product causing the problem, or which team is responsible for handling the reported issue. Automating this process can have a big impact on productivity and enables the support teams to focus on helping their users. As a running example, we'll use the GitHub issues associated with a popular open source project: 🤗 Transformers! Let's now take a look at what information is contained in these issues, how to frame the task, and how to get the data.

The methods presented in this chapter work well for text classification, but other techniques such as data augmentation may be necessary for tackling more complex tasks like named entity recognition, question answering, or summarization.

Building a GitHub Issues Tagger

If you navigate to the Issues tab of the 🤗 Transformers repository, you'll find issues like the one shown in Figure 9-2, which contains a title, a description, and a set of tags or labels that characterize the issue. This suggests a natural way to frame the supervised learning task: given a title and description of an issue, predict one or more labels. Since each issue can be assigned a variable number of labels, this means we are dealing with a *multilabel text classification* problem. This is usually more challenging than the multiclass problem that we encountered in Chapter 2, where each tweet was assigned to only one emotion.

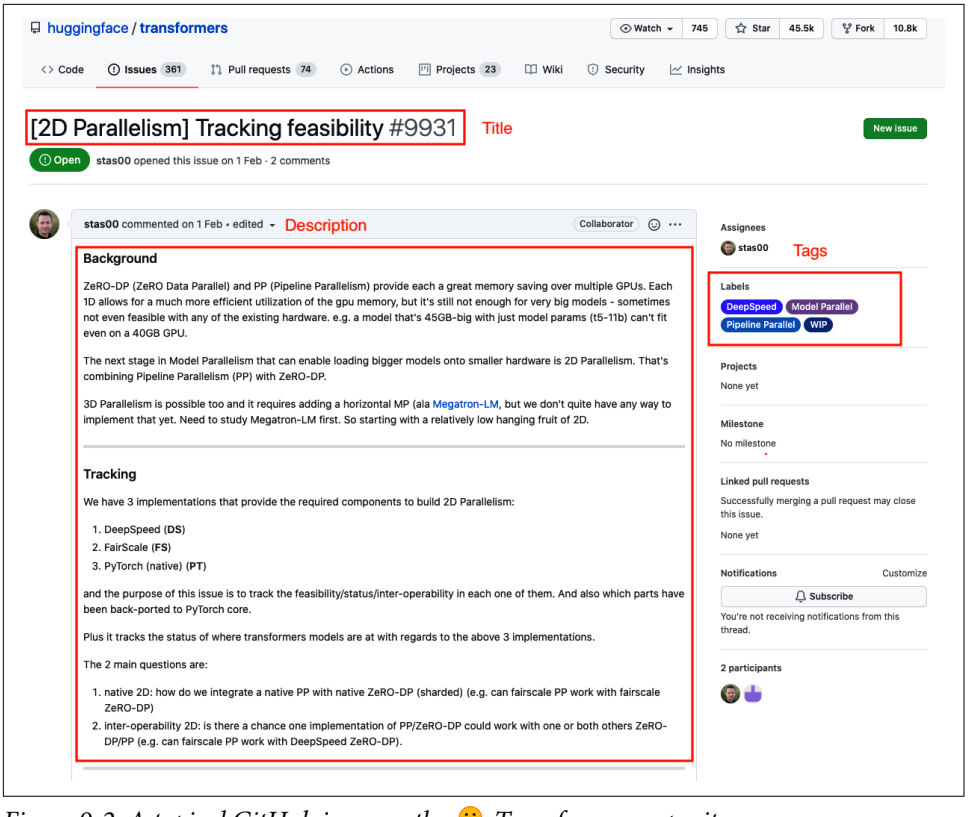


Figure 9-2. A typical GitHub issue on the 🤗 Transformers repository

Now that we've seen what the GitHub issues look like, let's see how we can download them to create our dataset.

Getting the Data

To grab all the repository's issues, we'll use the GitHub REST API to poll the `Issues` endpoint. This endpoint returns a list of JSON objects, with each containing a large number of fields about the issue at hand, including its state (open or closed), who opened the issue, as well as the title, body, and labels we saw in Figure 9-2.

Since it takes a while to fetch all the issues, we've included a *github-issues-transformers.jsonl* file in this book's GitHub repository, along with a `fetch_issues()` function that you can use to download them yourself.

The GitHub REST API treats pull requests as issues, so our dataset contains a mix of both. To keep things simple, we'll develop our classifier for both types of issue, although in practice you might consider building two separate classifiers to have more fine-grained control over the model's performance.

Now that we know how to grab the data, let's take a look at cleaning it up.

Preparing the Data

Once we've downloaded all the issues, we can load them using Pandas:

```
import pandas as pd

dataset_url = "https://git.io/nlp-with-transformers"
df_issues = pd.read_json(dataset_url, lines=True)
print(f"DataFrame shape: {df_issues.shape}")
```

```
DataFrame shape: (9930, 26)
```

There are almost 10,000 issues in our dataset, and by looking at a single row we can see that the information retrieved from the GitHub API contains many fields such as URLs, IDs, dates, users, title, body, as well as labels:

```
cols = ["url", "id", "title", "user", "labels", "state", "created_at", "body"]
df_issues.loc[2, cols].to_frame()
```

	2
url	https://api.github.com/repos/huggingface/trans...
id	849529761
title	[DeepSpeed] ZeRO stage 3 integration: getting ...
user	{'login': 'stas00', 'id': 10676103, 'node_id':...
labels	[{'id': 2659267025, 'node_id': 'MDU6TGFiZWwyNj...
state	open
created_at	2021-04-02 23:40:42
body	**[This is not yet alive, preparing for the re...

The `labels` column is the thing that we're interested in, and each row contains a list of JSON objects with metadata about each label:

```
[
   {
      "id":2659267025,
      "node_id":"MDU6TGFiZWwyNjU5MjY3MDI1",
      "url":"https://api.github.com/repos/huggingface...",
      "name":"DeepSpeed",
      "color":"4D34F7",
```

```
        "default":false,
        "description":""
    }
]
```

For our purposes, we're only interested in the `name` field of each label object, so let's overwrite the `labels` column with just the label names:

```
df_issues["labels"] = (df_issues["labels"]
                       .apply(lambda x: [meta["name"] for meta in x]))
df_issues[["labels"]].head()
```

	labels
0	[]
1	[]
2	[DeepSpeed]
3	[]
4	[]

Now each row in the `labels` column is a list of GitHub labels, so we can compute the length of each row to find the number of labels per issue:

```
df_issues["labels"].apply(lambda x : len(x)).value_counts().to_frame().T
```

	0	1	2	3	4	5
labels	6440	3057	305	100	25	3

This shows that the majority of issues have zero or one label, and much fewer have more than one. Next let's take a look at the top 10 most frequent labels in the dataset. In Pandas we can do this by "exploding" the `labels` column so that each label in the list becomes a row, and then simply counting the occurrences of each label:

```
df_counts = df_issues["labels"].explode().value_counts()
print(f"Number of labels: {len(df_counts)}")
# Display the top-8 label categories
df_counts.to_frame().head(8).T
```

```
Number of labels: 65
```

	wontfix	model card	Core: Tokenization	New model	Core: Modeling	Help wanted	Good First Issue	Usage
labels	2284	649	106	98	64	52	50	46

We can see that there are 65 unique labels in the dataset and that the classes are very imbalanced, with `wontfix` and `model card` being the most common labels. To make the classification problem more tractable, we'll focus on building a tagger for a subset

of the labels. For example, some labels, such as `Good First Issue` or `Help Wanted`, are potentially very difficult to predict from the issue's description, while others, such as `model card`, could be classified with a simple rule that detects when a model card is added on the Hugging Face Hub.

The following code filters the dataset for the subset of labels that we'll work with, along with a standardization of the names to make them easier to read:

```
label_map = {"Core: Tokenization": "tokenization",
             "New model": "new model",
             "Core: Modeling": "model training",
             "Usage": "usage",
             "Core: Pipeline": "pipeline",
             "TensorFlow": "tensorflow or tf",
             "PyTorch": "pytorch",
             "Examples": "examples",
             "Documentation": "documentation"}

def filter_labels(x):
    return [label_map[label] for label in x if label in label_map]

df_issues["labels"] = df_issues["labels"].apply(filter_labels)
all_labels = list(label_map.values())
```

Now let's look at the distribution of the new labels:

```
df_counts = df_issues["labels"].explode().value_counts()
df_counts.to_frame().T
```

	tokenization	new model	model training	usage	pipeline	tensorflow or tf	pytorch	documentation	examples
labels	106	98	64	46	42	41	37	28	24

Later in this chapter we'll find it useful to treat the unlabeled issues as a separate training split, so let's create a new column that indicates whether the issue is unlabeled or not:

```
df_issues["split"] = "unlabeled"
mask = df_issues["labels"].apply(lambda x: len(x)) > 0
df_issues.loc[mask, "split"] = "labeled"
df_issues["split"].value_counts().to_frame()
```

	split
unlabeled	9489
labeled	441

Let's now take a look at an example:

```
for column in ["title", "body", "labels"]:
    print(f"{column}: {df_issues[column].iloc[26][:500]}\n")
```

```
title: Add new CANINE model

body: # ⭐ New model addition

## Model description

Google recently proposed a new **C**haracter **A**rchitecture with **N**o
 tokenization **I**n **N**eural **E**ncoders architecture (CANINE). Not only
 the title is exciting:

Pipelined NLP systems have largely been superseded by end-to-end neural
 modeling, yet nearly all commonly-used models still require an explicit
 tokenization step. While recent tokenization approaches based on data-derived
 subword lexicons are less brittle than manually en

labels: ['new model']
```

In this example a new model architecture is proposed, so the `new model` tag makes sense. We can also see that the `title` contains information that will be useful for our classifier, so let's concatenate it with the issue's description in the `body` field:

```
df_issues["text"] = (df_issues
                     .apply(lambda x: x["title"] + "\n\n" + x["body"], axis=1))
```

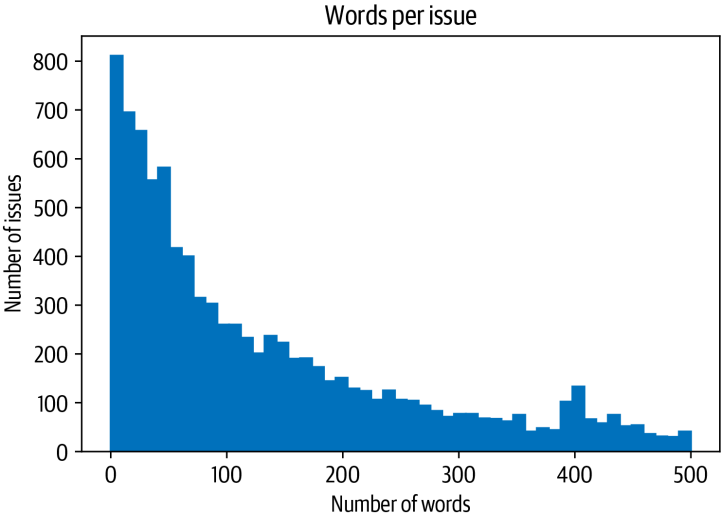
Before we look at the rest of the data, let's check for any duplicates in the data and drop them with the `drop_duplicates()` method:

```
len_before = len(df_issues)
df_issues = df_issues.drop_duplicates(subset="text")
print(f"Removed {(len_before-len(df_issues))/len_before:.2%} duplicates.")
```

```
Removed 1.88% duplicates.
```

We can see that there were a few duplicate issues in our dataset, but they only represented a small percentage. As we've done in other chapters, it's also a good idea to have a quick look at the number of words in our texts to see if we'll lose much information when we truncate to each model's context size:

```
import numpy as np
import matplotlib.pyplot as plt

(df_issues["text"].str.split().apply(len)
 .hist(bins=np.linspace(0, 500, 50), grid=False, edgecolor="C0"))
plt.title("Words per issue")
plt.xlabel("Number of words")
plt.ylabel("Number of issues")
plt.show()
```



The distribution has the long tail characteristic of many text datasets. Most of the texts are fairly short, but there are also issues with more than 500 words. It is common to have some very long issues, especially when error messages and code snippets are posted along with them. Given that most transformer models have a context size of 512 tokens or larger, truncating a handful of long issues is not likely to affect the overall performance. Now that we've explored and cleaned up our dataset, the final thing to do is define our training and validation sets to benchmark our classifiers. Let's take a look at how to do this.

Creating Training Sets

Creating training and validation sets is a bit trickier for multlilabel problems because there is no guaranteed balance for all labels. However, it can be approximated, and we can use the Scikit-multilearn library, which is specifically set up for this purpose. The first thing we need to do is transform our set of labels, like `pytorch` and `tokenization`, into a format that the model can process. Here we can use Scikit-learn's `MultiLabelBinarizer` class, which takes a list of label names and creates a vector with zeros for absent labels and ones for present labels. We can test this by fitting `MultiLabelBinarizer` on `all_labels` to learn the mapping from label name to ID as follows:

```
from sklearn.preprocessing import MultiLabelBinarizer

mlb = MultiLabelBinarizer()
mlb.fit([all_labels])
mlb.transform([["tokenization", "new model"], ["pytorch"]])
```

```
array([[0, 0, 0, 1, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0]])
```

In this simple example we can see the first row has two ones corresponding to the `tokenization` and `new model` labels, while the second row has just one hit with `pytorch`.

To create the splits we can use the `iterative_train_test_split()` function from Scikit-multilearn, which creates the train/test splits iteratively to achieve balanced labels. We wrap it in a function that we can apply to `DataFrames`. Since the function expects a two-dimensional feature matrix, we need to add a dimension to the possible indices before making the split:

```
from skmultilearn.model_selection import iterative_train_test_split

def balanced_split(df, test_size=0.5):
    ind = np.expand_dims(np.arange(len(df)), axis=1)
    labels = mlb.transform(df["labels"])
    ind_train, _, ind_test, _ = iterative_train_test_split(ind, labels,
                                                           test_size)
    return df.iloc[ind_train[:, 0]], df.iloc[ind_test[:,0]]
```

Armed with the `balanced_split()` function, we can split the data into supervised and unsupervised datasets, and then create balanced training, validation, and test sets for the supervised part:

```
from sklearn.model_selection import train_test_split

df_clean = df_issues[["text", "labels", "split"]].reset_index(drop=True).copy()
df_unsup = df_clean.loc[df_clean["split"] == "unlabeled", ["text", "labels"]]
df_sup = df_clean.loc[df_clean["split"] == "labeled", ["text", "labels"]]

np.random.seed(0)
df_train, df_tmp = balanced_split(df_sup, test_size=0.5)
df_valid, df_test = balanced_split(df_tmp, test_size=0.5)
```

Finally, let's create a `DatasetDict` with all the splits so that we can easily tokenize the dataset and integrate with the `Trainer`. Here we'll use the nifty `from_pandas()` method to load each split directly from the corresponding Pandas `DataFrame`:

```
from datasets import Dataset, DatasetDict

ds = DatasetDict({
    "train": Dataset.from_pandas(df_train.reset_index(drop=True)),
    "valid": Dataset.from_pandas(df_valid.reset_index(drop=True)),
    "test": Dataset.from_pandas(df_test.reset_index(drop=True)),
    "unsup": Dataset.from_pandas(df_unsup.reset_index(drop=True))})
```

This looks good, so the last thing to do is to create some training slices so that we can evaluate the performance of each classifier as a function of the training set size.

Creating Training Slices

The dataset has the two characteristics that we'd like to investigate in this chapter: sparse labeled data and multilabel classification. The training set consists of only 220 examples to train with, which is certainly a challenge even with transfer learning. To drill down into how each method in this chapter performs with little labeled data, we'll also create slices of the training data with even fewer samples. We can then plot the number of samples against the performance and investigate various regimes. We'll start with only eight samples per label and build up until the slice covers the full training set using the `iterative_train_test_split()` function:

```
np.random.seed(0)
all_indices = np.expand_dims(list(range(len(ds["train"]))), axis=1)
indices_pool = all_indices
labels = mlb.transform(ds["train"]["labels"])
train_samples = [8, 16, 32, 64, 128]
train_slices, last_k = [], 0

for i, k in enumerate(train_samples):
    # Split off samples necessary to fill the gap to the next split size
    indices_pool, labels, new_slice, _ = iterative_train_test_split(
        indices_pool, labels, (k-last_k)/len(labels))
    last_k = k
    if i==0: train_slices.append(new_slice)
    else: train_slices.append(np.concatenate((train_slices[-1], new_slice)))

# Add full dataset as last slice
train_slices.append(all_indices), train_samples.append(len(ds["train"]))
train_slices = [np.squeeze(train_slice) for train_slice in train_slices]
```

Note that this iterative approach only approximately splits the samples to the desired size, since it is not always possible to find a balanced split at a given split size:

```
print("Target split sizes:")
print(train_samples)
print("Actual split sizes:")
print([len(x) for x in train_slices])
```

```
Target split sizes:
[8, 16, 32, 64, 128, 223]
Actual split sizes:
[10, 19, 36, 68, 134, 223]
```

We'll use the specified split sizes as the labels for the following plots. Great, we've finally prepared our dataset into training splits—let's next take a look at training a strong baseline model!

Implementing a Naive Bayesline

Whenever you start a new NLP project, it's always a good idea to implement a set of strong baselines. There are two main reasons for this:

1. A baseline based on regular expressions, handcrafted rules, or a very simple model might already work really well to solve the problem. In these cases, there is no reason to bring out big guns like transformers, which are generally more complex to deploy and maintain in production environments.
2. The baselines provide quick checks as you explore more complex models. For example, suppose you train BERT-large and get an accuracy of 80% on your validation set. You might write it off as a hard dataset and call it a day. But what if you knew that a simple classifier like logistic regression gets 95% accuracy? That would raise your suspicions and prompt you to debug your model.

So let's start our analysis by training a baseline model. For text classification, a great baseline is a *Naive Bayes classifier* as it is very simple, quick to train, and fairly robust to perturbations in the inputs. The Scikit-learn implementation of Naive Bayes does not support multilabel classification out of the box, but fortunately we can again use the Scikit-multilearn library to cast the problem as a one-versus-rest classification task where we train L binary classifiers for L labels. First, let's use a multilabel binarizer to create a new `label_ids` column in our training sets. We can use the `map()` function to take care of all the processing in one go:

```
def prepare_labels(batch):
    batch["label_ids"] = mlb.transform(batch["labels"])
    return batch

ds = ds.map(prepare_labels, batched=True)
```

To measure the performance of our classifiers, we'll use the micro and macro F_1-scores, where the former tracks performance on the frequent labels and the latter on all labels disregarding the frequency. Since we'll be evaluating each model across different-sized training splits, let's create a `defaultdict` with a list to store the scores per split:

```
from collections import defaultdict

macro_scores, micro_scores = defaultdict(list), defaultdict(list)
```

Now we're finally ready to train our baseline! Here's the code to train the model and evaluate our classifier across increasing training set sizes:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report
from skmultilearn.problem_transform import BinaryRelevance
from sklearn.feature_extraction.text import CountVectorizer
```

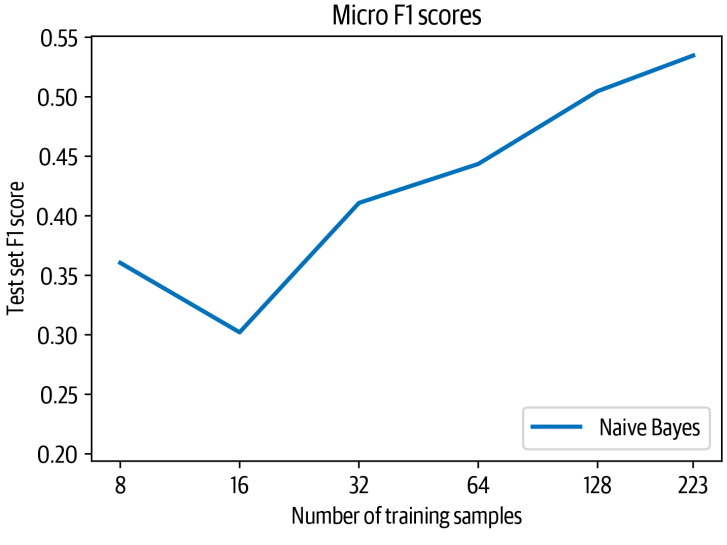
```
for train_slice in train_slices:
    # Get training slice and test data
    ds_train_sample = ds["train"].select(train_slice)
    y_train = np.array(ds_train_sample["label_ids"])
    y_test = np.array(ds["test"]["label_ids"])
    # Use a simple count vectorizer to encode our texts as token counts
    count_vect = CountVectorizer()
    X_train_counts = count_vect.fit_transform(ds_train_sample["text"])
    X_test_counts = count_vect.transform(ds["test"]["text"])
    # Create and train our model!
    classifier = BinaryRelevance(classifier=MultinomialNB())
    classifier.fit(X_train_counts, y_train)
    # Generate predictions and evaluate
    y_pred_test = classifier.predict(X_test_counts)
    clf_report = classification_report(
        y_test, y_pred_test, target_names=mlb.classes_, zero_division=0,
        output_dict=True)
    # Store metrics
    macro_scores["Naive Bayes"].append(clf_report["macro avg"]["f1-score"])
    micro_scores["Naive Bayes"].append(clf_report["micro avg"]["f1-score"])
```

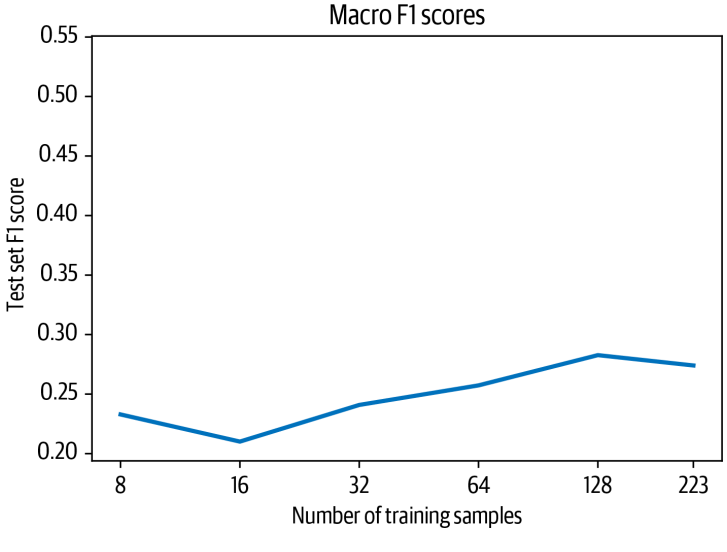
There's quite a lot going on in this block of code, so let's unpack it. First, we get the training slice and encode the labels. Then we use a count vectorizer to encode the texts by simply creating a vector of the size of the vocabulary where each entry corresponds to the frequency with which a token appeared in the text. This is called a *bag-of-words* approach, since all information on the order of the words is lost. Then we train the classifier and use the predictions on the test set to get the micro and macro F_1-scores via the classification report.

With the following helper function we can plot the results of this experiment:

```
import matplotlib.pyplot as plt

def plot_metrics(micro_scores, macro_scores, sample_sizes, current_model):
    fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(10, 4), sharey=True)

    for run in micro_scores.keys():
        if run == current_model:
            ax0.plot(sample_sizes, micro_scores[run], label=run, linewidth=2)
            ax1.plot(sample_sizes, macro_scores[run], label=run, linewidth=2)
        else:
            ax0.plot(sample_sizes, micro_scores[run], label=run,
                     linestyle="dashed")
            ax1.plot(sample_sizes, macro_scores[run], label=run,
                     linestyle="dashed")

    ax0.set_title("Micro F1 scores")
    ax1.set_title("Macro F1 scores")
    ax0.set_ylabel("Test set F1 score")
    ax0.legend(loc="lower right")
    for ax in [ax0, ax1]:
```

```
            ax.set_xlabel("Number of training samples")
            ax.set_xscale("log")
            ax.set_xticks(sample_sizes)
            ax.set_xticklabels(sample_sizes)
            ax.minorticks_off()
        plt.tight_layout()
        plt.show()

plot_metrics(micro_scores, macro_scores, train_samples, "Naive Bayes")
```





Note that we plot the number of samples on a logarithmic scale. From the figure we can see that the micro and macro F_1-scores both improve as we increase the number of training samples. With so few samples to train on, the results are also slightly noisy since each slice can have a different class distribution. Nevertheless, what's important here is the trend, so let's now see how these results fare against transformer-based approaches!

Working with No Labeled Data

The first technique that we'll consider is *zero-shot classification*, which is suitable in settings where you have no labeled data at all. This is surprisingly common in industry, and might occur because there is no historic data with labels or because acquiring the labels for the data is difficult. We will cheat a bit in this section since we will still use the test data to measure the performance, but we will not use any data to train the model (otherwise the comparison to the following approaches would be difficult).

The goal of zero-shot classification is to make use of a pretrained model without any additional fine-tuning on your task-specific corpus. To get a better idea of how this could work, recall that language models like BERT are pretrained to predict masked tokens in text on thousands of books and large Wikipedia dumps. To successfully predict a missing token, the model needs to be aware of the topic in the context. We can try to trick the model into classifying a document for us by providing a sentence like:

> "This section was about the topic [MASK]."

The model should then give a reasonable suggestion for the document's topic, since this is a natural text to occur in the dataset.[2]

Let's illustrate this further with the following toy problem: suppose you have two children, and one of them likes movies with cars while the other enjoys movies with animals better. Unfortunately, they have already seen all the ones you know, so you want to build a function that tells you what topic a new movie is about. Naturally, you turn to transformers for this task. The first thing to try is to load BERT-base in the `fill-mask` pipeline, which uses the masked language model to predict the content of the masked tokens:

```
from transformers import pipeline

pipe = pipeline("fill-mask", model="bert-base-uncased")
```

2 We thank Joe Davison for suggesting this approach to us.

Next, let's construct a little movie description and add a prompt to it with a masked word. The goal of the prompt is to guide the model to help us make a classification. The `fill-mask` pipeline returns the most likely tokens to fill in the masked spot:

```
movie_desc = "The main characters of the movie madacascar \
are a lion, a zebra, a giraffe, and a hippo. "
prompt = "The movie is about [MASK]."

output = pipe(movie_desc + prompt)
for element in output:
    print(f"Token {element['token_str']}:\t{element['score']:.3f}%")
```

```
Token animals:  0.103%
Token lions:    0.066%
Token birds:    0.025%
Token love:     0.015%
Token hunting:  0.013%
```

Clearly, the model predicts only tokens that are related to animals. We can also turn this around, and instead of getting the most likely tokens we can query the pipeline for the probability of a few given tokens. For this task we might choose `cars` and `animals`, so we can pass them to the pipeline as targets:

```
output = pipe(movie_desc + prompt, targets=["animals", "cars"])
for element in output:
    print(f"Token {element['token_str']}:\t{element['score']:.3f}%")
```

```
Token animals:  0.103%
Token cars:     0.001%
```

Unsurprisingly, the predicted probability for the token `cars` is much smaller than for `animals`. Let's see if this also works for a description that is closer to cars:

```
movie_desc = "In the movie transformers aliens \
can morph into a wide range of vehicles."

output = pipe(movie_desc + prompt, targets=["animals", "cars"])
for element in output:
    print(f"Token {element['token_str']}:\t{element['score']:.3f}%")
```

```
Token cars:     0.139%
Token animals:  0.006%
```

It does! This is only a simple example, and if we want to make sure it works well we should test it thoroughly, but it illustrates the key idea of many approaches discussed in this chapter: find a way to adapt a pretrained model for another task without training it. In this case we set up a prompt with a mask in such a way that we can use a masked language model directly for classification. Let's see if we can do better by adapting a model that has been fine-tuned on a task that's closer to text classification: *natural language inference* (NLI).

Using the masked language model for classification is a nice trick, but we can do better still by using a model that has been trained on a task that is closer to classification. There is a neat proxy task called *text entailment* that fits the bill. In text entailment, the model needs to determine whether two text passages are likely to follow or contradict each other. Models are typically trained to detect entailments and contradictions with datasets such as Multi-Genre NLI Corpus (MNLI) or Cross-Lingual NLI Corpus (XNLI).[3]

Each sample in these datasets is composed of three parts: a premise, a hypothesis, and a label, which can be one of `entailment`, `neutral`, or `contradiction`. The `entailment` label is assigned when the hypothesis text is necessarily true under the premise. The `contradiction` label is used when the hypothesis is necessarily false or inappropriate under the premise. If neither of these cases applies, then the `neutral` label is assigned. See Table 9-1 for examples of each.

Table 9-1. The three classes in the MLNI dataset

Premise	Hypothesis	Label
His favourite color is blue.	He is into heavy metal music.	`neutral`
She finds the joke hilarious.	She thinks the joke is not funny at all.	`contradiction`
The house was recently built.	The house is new.	`entailment`

Now, it turns out that we can hijack a model trained on the MNLI dataset to build a classifier without needing any labels at all! The key idea is to treat the text we wish to classify as the premise, and then formulate the hypothesis as:

> "This example is about {label}."

where we insert the class name for the label. The entailment score then tells us how likely that premise is to be about that topic, and we can run this for any number of classes sequentially. The downside of this approach is that we need to execute a forward pass for each class, which makes it less efficient than a standard classifier. Another slightly tricky aspect is that the choice of label names can have a large impact on the accuracy, and choosing labels with semantic meaning is generally the best approach. For example, if the label is simply `Class 1`, the model has no hint what this might mean and whether this constitutes a contradiction or entailment.

🤗 Transformers has an MNLI model for zero-shot classification built in. We can initialize it via a pipeline as follows:

3 A. Williams, N. Nangia, and S.R. Bowman, "A Broad-Coverage Challenge Corpus for Sentence Understanding Through Inference", (2018); A. Conneau et al., "XNLI: Evaluating Cross-Lingual Sentence Representations", (2018).

```
from transformers import pipeline

pipe = pipeline("zero-shot-classification", device=0)
```

The setting `device=0` makes sure that the model runs on the GPU instead of the default CPU to speed up inference. To classify a text, we simply need to pass it to the pipeline along with the label names. In addition, we can set `multi_label=True` to ensure that all the scores are returned and not only the maximum for single-label classification:

```
sample = ds["train"][0]
print(f"Labels: {sample['labels']}")
output = pipe(sample["text"], all_labels, multi_label=True)
print(output["sequence"][:400])
print("\nPredictions:")

for label, score in zip(output["labels"], output["scores"]):
    print(f"{label}, {score:.2f}")
```

```
Labels: ['new model']
Add new CANINE model

# ⭐ New model addition

## Model description

Google recently proposed a new **C**haracter **A**rchitecture with **N**o
tokenization **I**n **N**eural **E**ncoders architecture (CANINE). Not only the
title is exciting:

> Pipelined NLP systems have largely been superseded by end-to-end neural
modeling, yet nearly all commonly-used models still require an explicit tokeni

Predictions:
new model, 0.98
tensorflow or tf, 0.37
examples, 0.34
usage, 0.30
pytorch, 0.25
documentation, 0.25
model training, 0.24
tokenization, 0.17
pipeline, 0.16
```

Since we are using a subword tokenizer, we can even pass code to the model! The tokenization might not be very efficient because only a small fraction of the pretraining dataset for the zero-shot pipeline consists of code snippets, but since code is also made up of a lot of natural words this is not a big issue. Also, the code block might contain important information, such as the framework (PyTorch or TensorFlow).

We can see that the model is very confident that this text is about a new model, but it also produces relatively high scores for the other labels. An important aspect for zero-shot classification is the domain we're operating in. The texts we are dealing with here are very technical and mostly about coding, which makes them quite different from the original text distribution in the MNLI dataset. Thus, it is not surprising that this is a challenging task for the model; it might work much better for some domains than others, depending on how close they are to the training data.

Let's write a function that feeds a single example through the zero-shot pipeline, and then scale it out to the whole validation set by running `map()`:

```
def zero_shot_pipeline(example):
    output = pipe(example["text"], all_labels, multi_label=True)
    example["predicted_labels"] = output["labels"]
    example["scores"] = output["scores"]
    return example

ds_zero_shot = ds["valid"].map(zero_shot_pipeline)
```

Now that we have our scores, the next step is to determine which set of labels should be assigned to each example. There are a few options we can experiment with:

- Define a threshold and select all labels above the threshold.
- Pick the top *k* labels with the *k* highest scores.

To help us determine which method is best, let's write a `get_preds()` function that applies one of the approaches to retrieve the predictions:

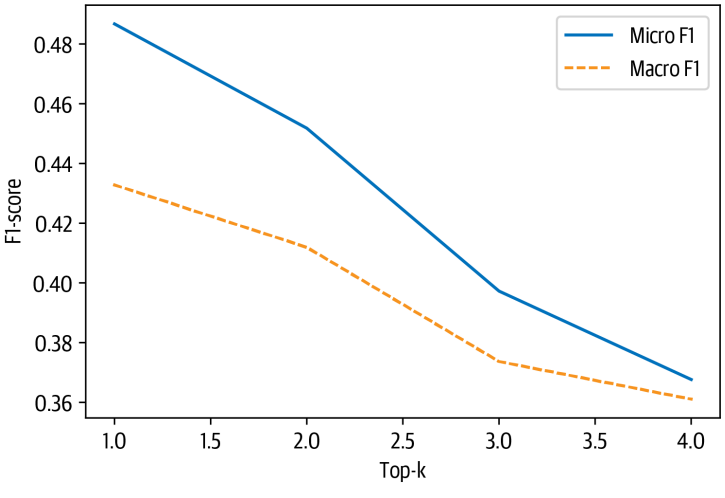
```
def get_preds(example, threshold=None, topk=None):
    preds = []
    if threshold:
        for label, score in zip(example["predicted_labels"], example["scores"]):
            if score >= threshold:
                preds.append(label)
    elif topk:
        for i in range(topk):
            preds.append(example["predicted_labels"][i])
    else:
        raise ValueError("Set either `threshold` or `topk`.")
    return {"pred_label_ids": list(np.squeeze(mlb.transform([preds])))}
```

Next, let's write a second function, `get_clf_report()`, that returns the Scikit-learn classification report from a dataset with the predicted labels:

```
def get_clf_report(ds):
    y_true = np.array(ds["label_ids"])
    y_pred = np.array(ds["pred_label_ids"])
    return classification_report(
        y_true, y_pred, target_names=mlb.classes_, zero_division=0,
        output_dict=True)
```

Armed with these two functions, let's start with the top-k method by increasing k for several values and then plotting the micro and macro F_1-scores across the validation set:

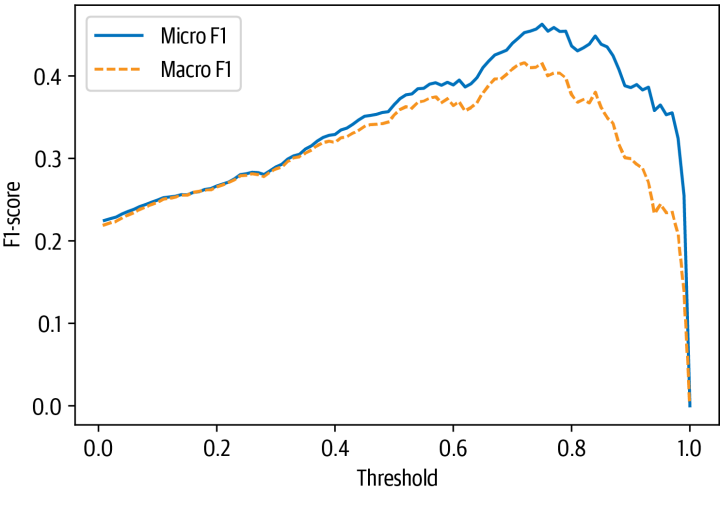
```
macros, micros = [], []
topks = [1, 2, 3, 4]
for topk in topks:
    ds_zero_shot = ds_zero_shot.map(get_preds, batched=False,
                                    fn_kwargs={'topk': topk})
    clf_report = get_clf_report(ds_zero_shot)
    micros.append(clf_report['micro avg']['f1-score'])
    macros.append(clf_report['macro avg']['f1-score'])

plt.plot(topks, micros, label='Micro F1')
plt.plot(topks, macros, label='Macro F1')
plt.xlabel("Top-k")
plt.ylabel("F1-score")
plt.legend(loc='best')
plt.show()
```



From the plot we can see that the best results are obtained by selecting the label with the highest score per example (top 1). This is perhaps not so surprising, given that most of the examples in our datasets have only one label. Let's now compare this against setting a threshold, so we can potentially predict more than one label per example:

```
macros, micros = [], []
thresholds = np.linspace(0.01, 1, 100)
for threshold in thresholds:
    ds_zero_shot = ds_zero_shot.map(get_preds,
                                    fn_kwargs={"threshold": threshold})
    clf_report = get_clf_report(ds_zero_shot)
```

```
        micros.append(clf_report["micro avg"]["f1-score"])
        macros.append(clf_report["macro avg"]["f1-score"])

plt.plot(thresholds, micros, label="Micro F1")
plt.plot(thresholds, macros, label="Macro F1")
plt.xlabel("Threshold")
plt.ylabel("F1-score")
plt.legend(loc="best")
plt.show()
```




```
best_t, best_micro = thresholds[np.argmax(micros)], np.max(micros)
print(f'Best threshold (micro): {best_t} with F1-score {best_micro:.2f}.')
best_t, best_macro = thresholds[np.argmax(macros)], np.max(macros)
print(f'Best threshold (micro): {best_t} with F1-score {best_macro:.2f}.')
```

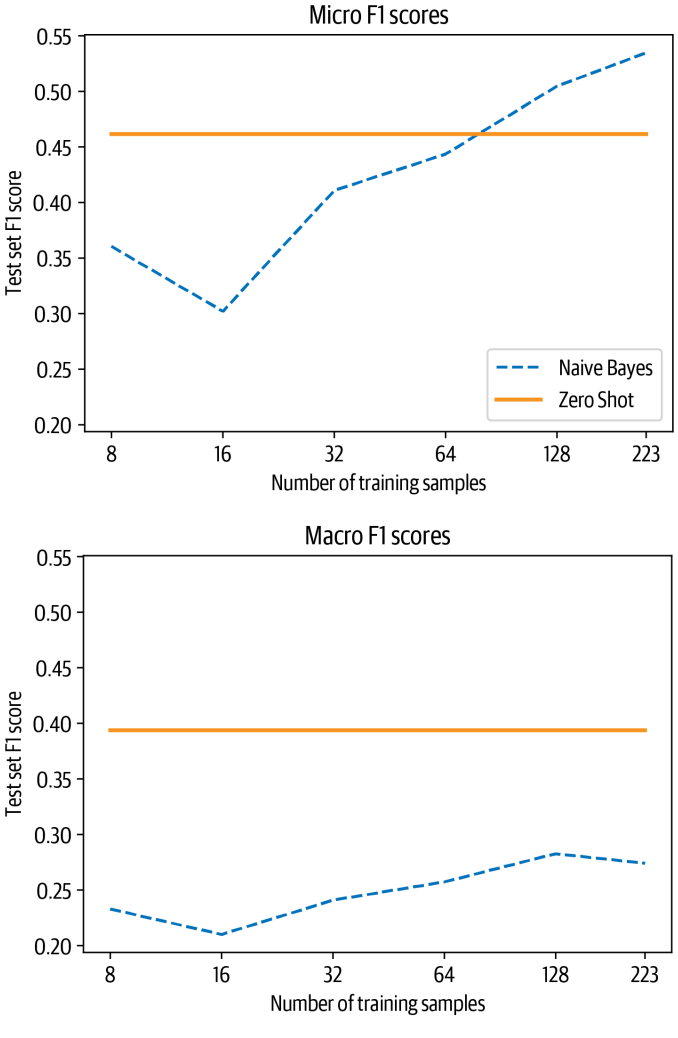
```
Best threshold (micro): 0.75 with F1-score 0.46.
Best threshold (micro): 0.72 with F1-score 0.42.
```

This approach fares somewhat worse than the top-1 results, but we can see the precision/recall trade-off clearly in this graph. If we set the threshold too low, then there are too many predictions, which leads to a low precision. If we set the threshold too high, then we will make hardly any predictions, which produces a low recall. From the plot we can see that a threshold value of around 0.8 is the sweet spot between the two.

Since the top-1 method performs best, let's use this to compare zero-shot classification against Naive Bayes on the test set:

```
ds_zero_shot = ds['test'].map(zero_shot_pipeline)
ds_zero_shot = ds_zero_shot.map(get_preds, fn_kwargs={'topk': 1})
clf_report = get_clf_report(ds_zero_shot)
for train_slice in train_slices:
```

```
        macro_scores['Zero Shot'].append(clf_report['macro avg']['f1-score'])
        micro_scores['Zero Shot'].append(clf_report['micro avg']['f1-score'])

plot_metrics(micro_scores, macro_scores, train_samples, "Zero Shot")
```



Comparing the zero-shot pipeline to the baseline, we observe two things:

1. If we have less than 50 labeled samples, the zero-shot pipeline handily outperforms the baseline.
2. Even above 50 samples, the performance of the zero-shot pipeline is superior when considering both the micro and macro F_1-scores. The results for the micro F_1-score tell us that the baseline performs well on the frequent classes, while the

zero-shot pipeline excels at those since it does not require any examples to learn from.

You might notice a slight paradox in this section: although we talk about dealing with no labels, we still use the validation and test sets. We use them to showcase different techniques and to make the results comparable between them. Even in a real use case, it makes sense to gather a handful of labeled examples to run some quick evaluations. The important point is that we did not adapt the parameters of the model with the data; instead, we just adapted some hyperparameters.

If you find it difficult to get good results on your own dataset, here are a few things you can do to improve the zero-shot pipeline:

- The way the pipeline works makes it very sensitive to the names of the labels. If the names don't make much sense or are not easily connected to the texts, the pipeline will likely perform poorly. Either try using different names or use several names in parallel and aggregate them in an extra step.
- Another thing you can improve is the form of the hypothesis. By default it is `hypothesis="This is example is about {}"`, but you can pass any other text to the pipeline. Depending on the use case, this might improve the performance.

Let's now turn to the regime where we have a few labeled examples we can use to train a model.

Working with a Few Labels

In most NLP projects, you'll have access to at least a few labeled examples. The labels might come directly from a client or cross-company team, or you might decide to just sit down and annotate a few examples yourself. Even for the previous approach, we needed a few labeled examples to evaluate how well the zero-shot approach worked. In this section, we'll have a look at how we can best leverage the few, precious labeled examples that we have. Let's start by looking at a technique known as data augmentation that can help us multiply the little labeled data that we have.

Data Augmentation

One simple but effective way to boost the performance of text classifiers on small datasets is to apply *data augmentation* techniques to generate new training examples from the existing ones. This is a common strategy in computer vision, where images are randomly perturbed without changing the meaning of the data (e.g., a slightly

rotated cat is still a cat). For text, data augmentation is somewhat trickier because perturbing the words or characters can completely change the meaning. For example, the two questions "Are elephants heavier than mice?" and "Are mice heavier than elephants?" differ by a single word swap, but have opposite answers. However, if the text consists of more than a few sentences (like our GitHub issues do), then the noise introduced by these types of transformations will generally not affect the label. In practice, there are two types of data augmentation techniques that are commonly used:

Back translation
: Take a text in the source language, translate it into one or more target languages using machine translation, and then translate it back to the source language. Back translation tends to works best for high-resource languages or corpora that don't contain too many domain-specific words.

Token perturbations
: Given a text from the training set, randomly choose and perform simple transformations like random synonym replacement, word insertion, swap, or deletion.[4]

Examples of these transformations are shown in Table 9-2. For a detailed list of other data augmentation techniques for NLP, we recommend reading Amit Chaudhary's blog post "A Visual Survey of Data Augmentation in NLP".

Table 9-2. Different types of data augmentation techniques for text

Augmentation	Sentence
None	Even if you defeat me Megatron, others will rise to defeat your tyranny
Synonym replace	Even if you kill me Megatron, others will prove to defeat your tyranny
Random insert	Even if you defeat me Megatron, others humanity will rise to defeat your tyranny
Random swap	You even if defeat me Megatron, others will rise defeat to tyranny your
Random delete	Even if you me Megatron, others to defeat tyranny
Back translate (German)	Even if you defeat me, others will rise up to defeat your tyranny

You can implement back translation using machine translation models like M2M100, while libraries like *NlpAug* and *TextAttack* provide various recipes for token perturbations. In this section, we'll focus on using synonym replacement as it's simple to implement and gets across the main idea behind data augmentation.

4 J. Wei and K. Zou, "EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks", (2019).

We'll use the `ContextualWordEmbsAug` augmenter from NlpAug to leverage the contextual word embeddings of DistilBERT for our synonym replacements. Let's start with a simple example:

```
from transformers import set_seed
import nlpaug.augmenter.word as naw

set_seed(3)
aug = naw.ContextualWordEmbsAug(model_path="distilbert-base-uncased",
                                device="cpu", action="substitute")

text = "Transformers are the most popular toys"
print(f"Original text: {text}")
print(f"Augmented text: {aug.augment(text)}")
```

```
Original text: Transformers are the most popular toys
Augmented text: transformers'the most popular toys
```

Here we can see how the word "are" has been replaced with an apostrophe to generate a new synthetic training example. We can wrap this augmentation in a simple function as follows:

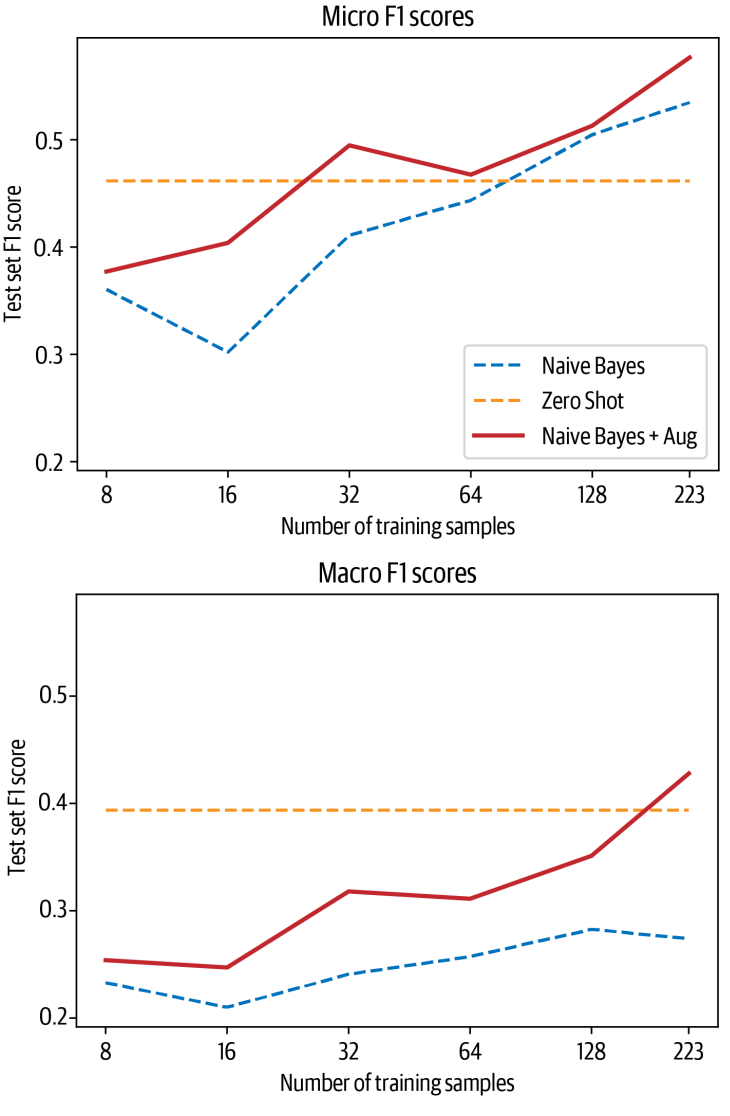
```
def augment_text(batch, transformations_per_example=1):
    text_aug, label_ids = [], []
    for text, labels in zip(batch["text"], batch["label_ids"]):
        text_aug += [text]
        label_ids += [labels]
        for _ in range(transformations_per_example):
            text_aug += [aug.augment(text)]
            label_ids += [labels]
    return {"text": text_aug, "label_ids": label_ids}
```

Now when we pass this function to the `map()` method, we can generate any number of new examples with the `transformations_per_example` argument. We can use this function in our code to train the Naive Bayes classifier by simply adding one line after we select the slice:

```
ds_train_sample = ds_train_sample.map(augment_text, batched=True,
    remove_columns=ds_train_sample.column_names).shuffle(seed=42)
```

Including this and rerunning the analysis produces the plot shown here:

```
plot_metrics(micro_scores, macro_scores, train_samples, "Naive Bayes + Aug")
```



From the figure, we can see that a small amount of data augmentation improves the F_1-score of the Naive Bayes classifier by around 5 points, and it overtakes the zero-shot pipeline for the macro scores once we have around 170 training samples. Let's now take a look at a method based on using the embeddings of large language models.

Using Embeddings as a Lookup Table

Large language models such as GPT-3 have been shown to be excellent at solving tasks with limited data. The reason is that these models learn useful representations of text that encode information across many dimensions, such as sentiment, topic, text structure, and more. For this reason, the embeddings of large language models can be used to develop a semantic search engine, find similar documents or comments, or even classify text.

In this section we'll create a text classifier that's modeled after the OpenAI API classification endpoint. The idea follows a three-step process:

1. Use the language model to embed all labeled texts.
2. Perform a nearest neighbor search over the stored embeddings.
3. Aggregate the labels of the nearest neighbors to get a prediction.

The process is illustrated in Figure 9-3, which shows how labeled data is embedded with a model and stored with the labels. When a new text needs to be classified it is embedded as well, and the label is given based on the labels of the nearest neighbors. It is important to calibrate the number of neighbors to be searched for, as too few might be noisy and too many might mix in neighboring groups.

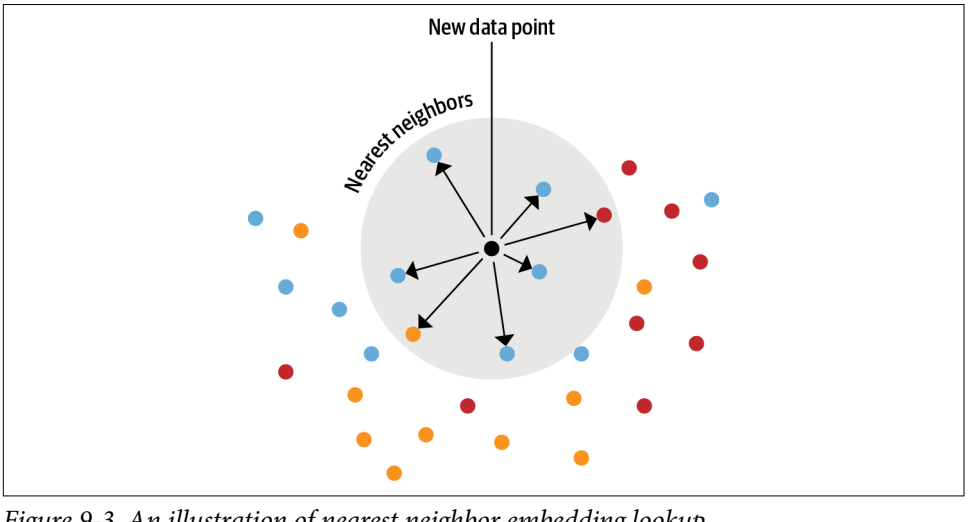


Figure 9-3. An illustration of nearest neighbor embedding lookup

The beauty of this approach is that no model fine-tuning is necessary to leverage the few available labeled data points. Instead, the main decision to make this approach work is to select an appropriate model that is ideally pretrained on a similar domain to your dataset.

Since GPT-3 is only available through the OpenAI API, we'll use GPT-2 to test the technique. Specifically, we'll use a variant of GPT-2 that was trained on Python code, which will hopefully capture some of the context contained in our GitHub issues.

Let's write a helper function that takes a list of texts and uses the model to create a single-vector representation for each text. One problem we have to deal with is that transformer models like GPT-2 will actually return one embedding vector per token. For example, given the sentence "I took my dog for a walk", we can expect several embedding vectors, one for each token. But what we really want is a single embedding vector for the whole sentence (or GitHub issue in our application). To deal with this, we can use a technique called *pooling*. One of the simplest pooling methods is to average the token embeddings, which is called *mean pooling*. With mean pooling, the only thing we need to watch out for is that we don't include padding tokens in the average, so we can use the attention mask to handle that.

To see how this works, let's load a GPT-2 tokenizer and model, define the mean pooling operation, and wrap the whole process in a simple `embed_text()` function:

```
import torch
from transformers import AutoTokenizer, AutoModel

model_ckpt = "miguelvictor/python-gpt2-large"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModel.from_pretrained(model_ckpt)

def mean_pooling(model_output, attention_mask):
    # Extract the token embeddings
    token_embeddings = model_output[0]
    # Compute the attention mask
    input_mask_expanded = (attention_mask
                           .unsqueeze(-1)
                           .expand(token_embeddings.size())
                           .float())
    # Sum the embeddings, but ignore masked tokens
    sum_embeddings = torch.sum(token_embeddings * input_mask_expanded, 1)
    sum_mask = torch.clamp(input_mask_expanded.sum(1), min=1e-9)
    # Return the average as a single vector
    return sum_embeddings / sum_mask

def embed_text(examples):
    inputs = tokenizer(examples["text"], padding=True, truncation=True,
                       max_length=128, return_tensors="pt")
    with torch.no_grad():
        model_output = model(**inputs)
    pooled_embeds = mean_pooling(model_output, inputs["attention_mask"])
    return {"embedding": pooled_embeds.cpu().numpy()}
```

Now we can get the embeddings for each split. Note that GPT-style models don't have a padding token, and therefore we need to add one before we can get the embeddings

in a batched fashion as implemented in the preceding code. We'll just recycle the end-of-string token for this purpose:

```
tokenizer.pad_token = tokenizer.eos_token
embs_train = ds["train"].map(embed_text, batched=True, batch_size=16)
embs_valid = ds["valid"].map(embed_text, batched=True, batch_size=16)
embs_test = ds["test"].map(embed_text, batched=True, batch_size=16)
```

Now that we have all the embeddings, we need to set up a system to search them. We could write a function that calculates, say, the cosine similarity between a new text embedding that we'll query and the existing embeddings in the training set. Alternatively, we can use a built-in structure of 🤗 Datasets called a *FAISS index*.[5] We already encountered FAISS in Chapter 7. You can think of this as a search engine for embeddings, and we'll have a closer look at how it works in a minute. We can use an existing field of the dataset to create a FAISS index with `add_faiss_index()`, or we can load new embeddings into the dataset with `add_faiss_index_from_external_arrays()`. Let's use the former function to add our training embeddings to the dataset as follows:

```
embs_train.add_faiss_index("embedding")
```

This created a new FAISS index called `embedding`. We can now perform a nearest neighbor lookup by calling the function `get_nearest_examples()`. It returns the closest neighbors as well as the matching score for each neighbor. We need to specify the query embedding as well as the number of nearest neighbors to retrieve. Let's give it a spin and have a look at the documents that are closest to an example:

```
i, k = 0, 3 # Select the first query and 3 nearest neighbors
rn, nl = "\r\n\r\n", "\n" # Used to remove newlines in text for compact display

query =  np.array(embs_valid[i]["embedding"], dtype=np.float32)
scores, samples = embs_train.get_nearest_examples("embedding", query, k=k)

print(f"QUERY LABELS: {embs_valid[i]['labels']}")
print(f"QUERY TEXT:\n{embs_valid[i]['text'][:200].replace(rn, nl)} [...]\n")
print("="*50)
print(f"Retrieved documents:")
for score, label, text in zip(scores, samples["labels"], samples["text"]):
    print("="*50)
    print(f"TEXT:\n{text[:200].replace(rn, nl)} [...]")
    print(f"SCORE: {score:.2f}")
    print(f"LABELS: {label}")
```

```
QUERY LABELS: ['new model']
QUERY TEXT:
Implementing efficient self attention in T5
```

5 J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs", (2017).

```
# ⭐ New model addition
My teammates and I (including @ice-americano) would like to use efficient self
attention methods such as Linformer, Performer and [...]

==================================================
Retrieved documents:
==================================================
TEXT:
Add Linformer model

# ⭐ New model addition
## Model description
### Linformer: Self-Attention with Linear Complexity
Paper published June 9th on ArXiv: https://arxiv.org/abs/2006.04768
La [...]
SCORE: 54.92
LABELS: ['new model']
==================================================
TEXT:
Add FAVOR+ / Performer attention

# ⭐ FAVOR+ / Performer attention addition
Are there any plans to add this new attention approximation block to
Transformers library?
## Model description
The n [...]
SCORE: 57.90
LABELS: ['new model']
==================================================
TEXT:
Implement DeLighT: Very Deep and Light-weight Transformers

# ⭐ New model addition
## Model description
DeLight, that delivers similar or better performance than transformer-based
models with sign [...]
SCORE: 60.12
LABELS: ['new model']
```

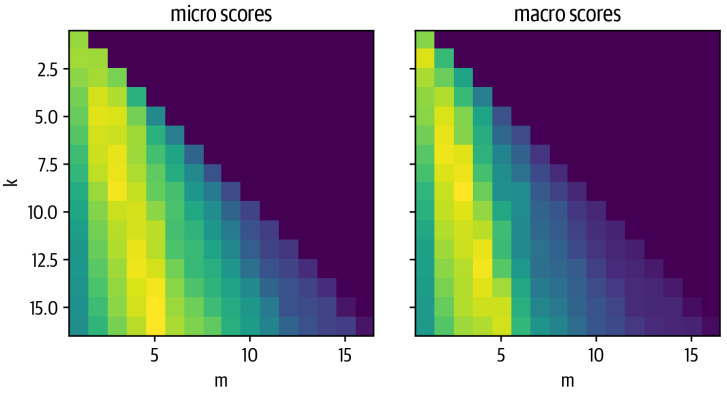
Nice! This is exactly what we hoped for: the three retrieved documents that we got via embedding lookup all have the same labels and we can already see from the titles that they are all very similar. The query as well as the retrieved documents revolve around adding new and efficient transformer models. The question remains, however, what is the best value for k? Similarly, how we should then aggregate the labels of the retrieved documents? Should we, for example, retrieve three documents and assign all labels that occurred at least twice? Or should we go for 20 and use all labels that appeared at least 5 times? Let's investigate this systematically: we'll try several values for k and then vary the threshold $m < k$ for label assignment with a helper function. We'll record the macro and micro performance for each setting so we can decide later which run performed best. Instead of looping over each sample in the validation set

we can make use of the function `get_nearest_examples_batch()`, which accepts a batch of queries:

```
def get_sample_preds(sample, m):
    return (np.sum(sample["label_ids"], axis=0) >= m).astype(int)

def find_best_k_m(ds_train, valid_queries, valid_labels, max_k=17):
    max_k = min(len(ds_train), max_k)
    perf_micro = np.zeros((max_k, max_k))
    perf_macro = np.zeros((max_k, max_k))
    for k in range(1, max_k):
        for m in range(1, k + 1):
            _, samples = ds_train.get_nearest_examples_batch("embedding",
                                                             valid_queries, k=k)
            y_pred = np.array([get_sample_preds(s, m) for s in samples])
            clf_report = classification_report(valid_labels, y_pred,
                target_names=mlb.classes_, zero_division=0, output_dict=True)
            perf_micro[k, m] = clf_report["micro avg"]["f1-score"]
            perf_macro[k, m] = clf_report["macro avg"]["f1-score"]
    return perf_micro, perf_macro
```

Let's check what the best values would be with all the training samples and visualize the scores for all *k* and *m* configurations:

```
valid_labels = np.array(embs_valid["label_ids"])
valid_queries = np.array(embs_valid["embedding"], dtype=np.float32)
perf_micro, perf_macro = find_best_k_m(embs_train, valid_queries, valid_labels)

fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(10, 3.5), sharey=True)
ax0.imshow(perf_micro)
ax1.imshow(perf_macro)

ax0.set_title("micro scores")
ax0.set_ylabel("k")
ax1.set_title("macro scores")
for ax in [ax0, ax1]:
    ax.set_xlim([0.5, 17 - 0.5])
    ax.set_ylim([17 - 0.5, 0.5])
    ax.set_xlabel("m")
plt.show()
```



From the plots we can see that there is a pattern: choosing m too large or small for a given k yields suboptimal results. The best performance is achieved when choosing a ratio of approximately $m/k = 1/3$. Let's see which k and m give the best result overall:

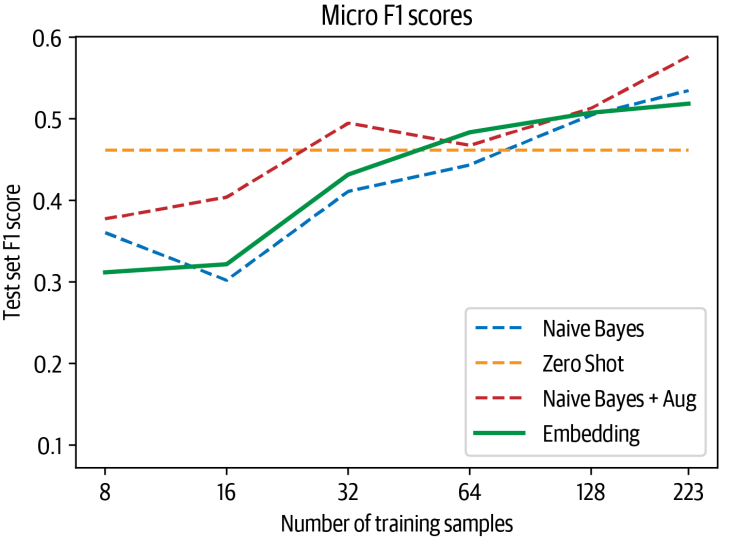
```
k, m = np.unravel_index(perf_micro.argmax(), perf_micro.shape)
print(f"Best k: {k}, best m: {m}")
```

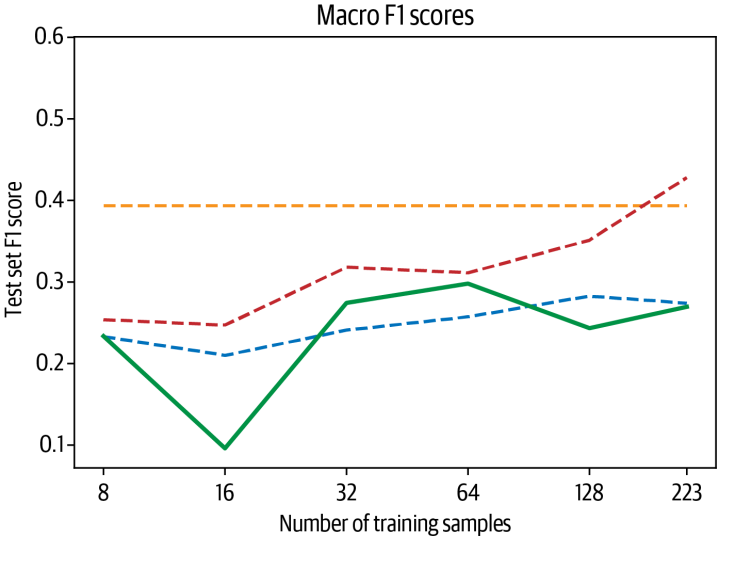
```
Best k: 15, best m: 5
```

The perfomance is best when we choose $k = 15$ and $m = 5$, or in other words when we retrieve the 15 nearest neighbors and then assign the labels that occurred at least 5 times. Now that we have a good method for finding the best values for the embedding lookup, we can play the same game as with the Naive Bayes classifier where we go through the slices of the training set and evaluate the performance. Before we can slice the dataset, we need to remove the index since we cannot slice a FAISS index like the dataset. The rest of the loops stay exactly the same, with the addition of using the validation set to get the best k and m values:

```
embs_train.drop_index("embedding")
test_labels = np.array(embs_test["label_ids"])
test_queries = np.array(embs_test["embedding"], dtype=np.float32)

for train_slice in train_slices:
    # Create a Faiss index from training slice
    embs_train_tmp = embs_train.select(train_slice)
    embs_train_tmp.add_faiss_index("embedding")
    # Get best k, m values with validation set
    perf_micro, _ = find_best_k_m(embs_train_tmp, valid_queries, valid_labels)
    k, m = np.unravel_index(perf_micro.argmax(), perf_micro.shape)
    # Get predictions on test set
    _, samples = embs_train_tmp.get_nearest_examples_batch("embedding",
                                                           test_queries,
                                                           k=int(k))
    y_pred = np.array([get_sample_preds(s, m) for s in samples])
    # Evaluate predictions
```

```
    clf_report = classification_report(test_labels, y_pred,
        target_names=mlb.classes_, zero_division=0, output_dict=True,)
    macro_scores["Embedding"].append(clf_report["macro avg"]["f1-score"])
    micro_scores["Embedding"].append(clf_report["micro avg"]["f1-score"])

plot_metrics(micro_scores, macro_scores, train_samples, "Embedding")
```





The embedding lookup is competitive on the micro scores with the previous approaches while just having two "learnable" parameters, k and m, but performs slightly worse on the macro scores.

Take these results with a grain of salt; which method works best strongly depends on the domain. The zero-shot pipeline's training data is quite different from the GitHub issues dataset we're using it on, which contains a lot of code that the model likely has not encountered much before. For a more common task such as sentiment analysis of reviews, the pipeline might work much better. Similarly, the embeddings' quality depends on the model and the data it was trained on. We tried half a dozen models, such as `sentence-transformers/stsb-roberta-large`, which was trained to give high-quality embeddings of sentences, and `microsoft/codebert-base` and `dbernsohn/roberta-python`, which were trained on code and documentation. For this specific use case, GPT-2 trained on Python code worked best.

Since you don't actually need to change anything in your code besides replacing the model checkpoint name to test another model, you can quickly try out a few models once you have the evaluation pipeline set up.

Let's now compare this simple embedding trick against simply fine-tuning a transformer on the limited data we have.

Efficient Similarity Search with FAISS

We first encountered FAISS in Chapter 7, where we used it to retrieve documents via the DPR embeddings. Here we'll explain briefly how the FAISS library works and why it is a powerful tool in the ML toolbox.

We are used to performing fast text queries on huge datasets such as Wikipedia or the web with search engines such as Google. When we move from text to embeddings, we would like to maintain that performance; however, the methods used to speed up text queries don't apply to embeddings.

To speed up text search we usually create an inverted index that maps terms to documents. An inverted index works like an index at the end of a book: each word is mapped to the pages (or in our case, document) it occurs in. When we later run a query we can quickly look up in which documents the search terms appear. This works well with discrete objects such as words, but does not work with continuous objects such as vectors. Each document likely has a unique vector, and therefore the index will never match with a new vector. Instead of looking for exact matches, we need to look for close or similar matches.

When we want to find the most similar vectors in a database to a query vector, in theory we need to compare the query vector to each of the n vectors in the database. For a small database such as we have in this chapter this is no problem, but if we

scaled this up to thousands or even million of entries we would need to wait a while for each query to be processed.

FAISS addresses this issue with several tricks. The main idea is to partition the dataset. If we only need to compare the query vector to a subset of the database, we can speed up the process significantly. But if we just randomly partition the dataset, how can we decide which partition to search, and what guarantees do we get for finding the most similar entries? Evidently, there must be a better solution: apply k-means clustering to the dataset! This clusters the embeddings into groups by similarity. Furthermore, for each group we get a centroid vector, which is the average of all members of the group (Figure 9-4).

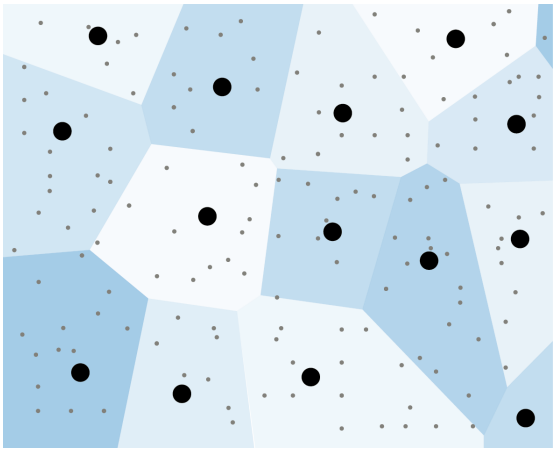
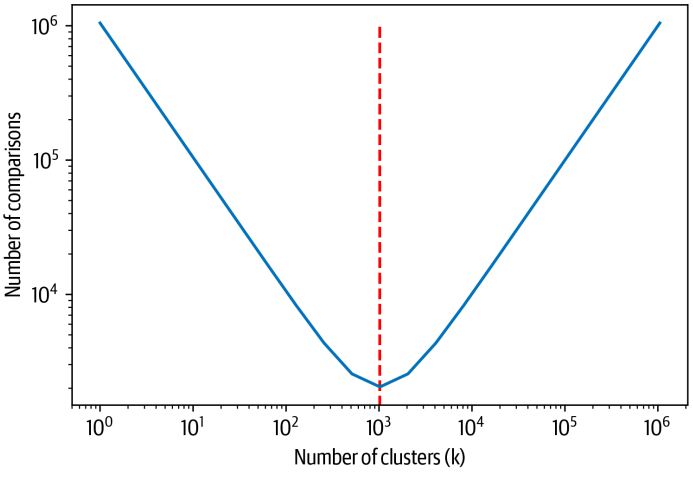


Figure 9-4. The structure of a FAISS index: the gray points represent data points added to the index, the bold black points are the cluster centers found via k-means clustering, and the colored areas represent the regions belonging to a cluster center

Given such a grouping, searching among n vectors is much easier: we first search across the k centroids for the one that is most similar to our query (k comparisons), and then we search within the group ($\frac{k}{n}$ elements to compare). This reduces the number of comparisons from n to $k + \frac{n}{k}$. So the question is, what is the best option for k? If it is too small, each group still contains many samples we need to compare against in the second step, and if k is too large there are many centroids we need to search through. Looking for the minimum of the function $f(k) = k + \frac{n}{k}$ with respect to k, we find $k = \sqrt{n}$. In fact, we can visualize this with the following graphic with $n = 2^{20}$.



In the plot you can see the number of comparisons as a function of the number of clusters. We are looking for the minimum of this function, where we need to do the least comparisons. We can see that the minimum is exactly where we expected to see it, at $\sqrt{2^{20}} = 2^{10} = 1,024$.

In addition to speeding up queries with partitioning, FAISS also allows you to utilize GPUs for a further speedup. If memory becomes a concern there are also several options to compress the vectors with advanced quantization schemes. If you want to use FAISS for your project, the repository has a simple guide for you to choose the right methods for your use case.

One of the largest projects to use FAISS was the creation of the CCMatrix corpus by Facebook. The authors used multilingual embeddings to find parallel sentences in different languages. This enormous corpus was subsequently used to train M2M100, a large machine translation model that is able to directly translate between any of 100 languages.

Fine-Tuning a Vanilla Transformer

If we have access to labeled data, we can also try to do the obvious thing: simply fine-tune a pretrained transformer model. In this section, we'll use the standard BERT checkpoint as a starting point. Later, we'll see the effect that fine-tuning the language model has on performance.

For many applications, starting with a pretrained BERT-like model is a good idea. However, if the domain of your corpus differs significantly from the pretraining corpus (which is usually Wikipedia), you should explore the many models that are available on the Hugging Face Hub. Chances are someone has already pretrained a model on your domain!

Let's start by loading the pretrained tokenizer, tokenizing our dataset, and getting rid of the columns we don't need for training and evaluation:

```
import torch
from transformers import (AutoTokenizer, AutoConfig,
                          AutoModelForSequenceClassification)

model_ckpt = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)

def tokenize(batch):
    return tokenizer(batch["text"], truncation=True, max_length=128)
ds_enc = ds.map(tokenize, batched=True)
ds_enc = ds_enc.remove_columns(['labels', 'text'])
```

The multilabel loss function expects the labels to be of type float, since it also allows for class probabilities instead of discrete labels. Therefore, we need to change the type of the column `label_ids`. Since changing the format of the column element-wise does not play well with Arrow's typed format, we'll do a little workaround. First, we create a new column with the labels. The format of that column is inferred from the first element. Then we delete the original column and rename the new one to take the place of the original one:

```
ds_enc.set_format("torch")
ds_enc = ds_enc.map(lambda x: {"label_ids_f": x["label_ids"].to(torch.float)},
                    remove_columns=["label_ids"])
ds_enc = ds_enc.rename_column("label_ids_f", "label_ids")
```

Since we are likely to quickly overfit the training data due to its limited size, we set `load_best_model_at_end=True` and choose the best model based on the micro F_1-score:

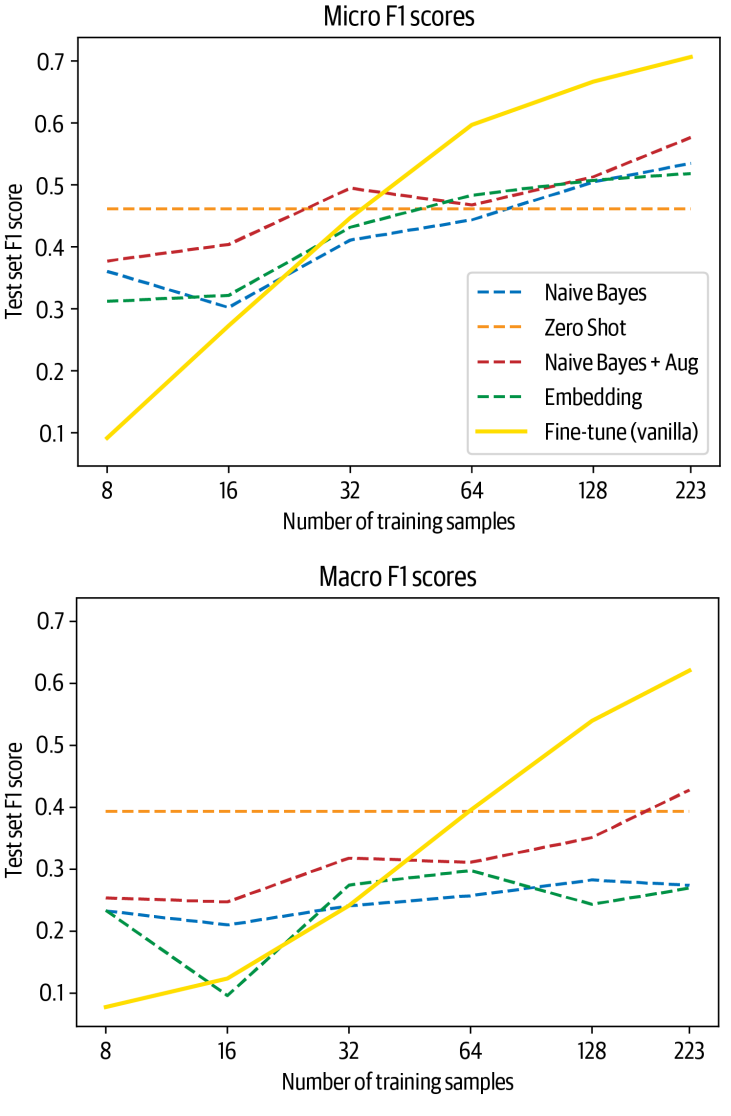
```
from transformers import Trainer, TrainingArguments

training_args_fine_tune = TrainingArguments(
    output_dir="./results", num_train_epochs=20, learning_rate=3e-5,
    lr_scheduler_type='constant', per_device_train_batch_size=4,
    per_device_eval_batch_size=32, weight_decay=0.0,
    evaluation_strategy="epoch", save_strategy="epoch",logging_strategy="epoch",
    load_best_model_at_end=True, metric_for_best_model='micro f1',
    save_total_limit=1, log_level='error')
```

We need the F_1-score to choose the best model, so we need to make sure it is calculated during the evaluation. Because the model returns the logits, we first need to normalize the predictions with a sigmoid function and can then binarize them with a simple threshold. Then we return the scores we are interested in from the classification report:

```
from scipy.special import expit as sigmoid

def compute_metrics(pred):
    y_true = pred.label_ids
    y_pred = sigmoid(pred.predictions)
    y_pred = (y_pred>0.5).astype(float)

    clf_dict = classification_report(y_true, y_pred, target_names=all_labels,
                                     zero_division=0, output_dict=True)
    return {"micro f1": clf_dict["micro avg"]["f1-score"],
            "macro f1": clf_dict["macro avg"]["f1-score"]}
```

Now we are ready to rumble! For each training set slice we train a classifier from scratch, load the best model at the end of the training loop, and store the results on the test set:

```
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels)
config.problem_type = "multi_label_classification"

for train_slice in train_slices:
    model = AutoModelForSequenceClassification.from_pretrained(model_ckpt,
                                                               config=config)
    trainer = Trainer(
        model=model, tokenizer=tokenizer,
        args=training_args_fine_tune,
        compute_metrics=compute_metrics,
        train_dataset=ds_enc["train"].select(train_slice),
        eval_dataset=ds_enc["valid"],)

    trainer.train()
    pred = trainer.predict(ds_enc["test"])
    metrics = compute_metrics(pred)
    macro_scores["Fine-tune (vanilla)"].append(metrics["macro f1"])
    micro_scores["Fine-tune (vanilla)"].append(metrics["micro f1"])

plot_metrics(micro_scores, macro_scores, train_samples, "Fine-tune (vanilla)")
```



First of all we see that simply fine-tuning a vanilla BERT model on the dataset leads to competitive results when we have access to around 64 examples. We also see that before this the behavior is a bit erratic, which is again due to training a model on a small sample where some labels can be unfavorably unbalanced. Before we make use of the unlabeled part of our dataset, let's take a quick look at another promising approach for using language models in the few-shot domain.

In-Context and Few-Shot Learning with Prompts

We saw earlier in this chapter that we can use a language model like BERT or GPT-2 and adapt it to a supervised task by using prompts and parsing the model's token predictions. This is different from the classic approach of adding a task-specific head and tuning the model parameters for the task. On the plus side, this approach does not require any training data, but on the negative side it seems we can't leverage labeled data if we have access to it. There is a middle ground that we can sometimes take advantage of called *in-context* or *few-shot learning*.

To illustrate the concept, consider an English to French translation task. In the zero-shot paradigm, we would construct a prompt that might look as follows:

```
prompt = """\
Translate English to French:
thanks =>
"""
```

This hopefully prompts the model to predict the tokens of the word "merci". We already saw when using GPT-2 for summarization in Chapter 6 that adding "TL;DR" to a text prompted the model to generate a summary without explicitly being trained to do this. An interesting finding of the GPT-3 paper was the ability of large language models to effectively learn from examples presented in the prompt—so, the previous translation example could be augmented with several English to German examples, which would make the model perform much better on this task.[6]

Furthermore, the authors found that the larger the models are scaled, the better they are at using the in-context examples, leading to significant performance boosts. Although GPT-3-sized models are challenging to use in production, this is an exciting emerging research field and people have built cool applications, such as a natural language shell where commands are entered in natural language and parsed by GPT-3 to shell commands.

An alternative approach to using labeled data is to create examples of the prompts and desired predictions and continue training the language model on these examples. A novel method called ADAPET uses such an approach and beats GPT-3 on a wide variety of tasks,[7] tuning the model with generated prompts. Recent work by Hugging Face researchers suggests that such an approach can be more data-efficient than fine-tuning a custom head.[8]

6 T. Brown et al., "Language Models Are Few-Shot Learners", (2020).

7 D. Tam et al., "Improving and Simplifying Pattern Exploiting Training", (2021).

8 T. Le Scao and A.M. Rush, "How Many Data Points Is a Prompt Worth?", (2021).

In this section we briefly looked at various ways to make good use of the few labeled examples that we have. Very often we also have access to a lot of unlabeled data in addition to the labeled examples; in the next section we'll discuss how to make good use of that.

Leveraging Unlabeled Data

Although having access to large volumes of high-quality labeled data is the best-case scenario to train a classifier, this does not mean that unlabeled data is worthless. Just think about the pretraining of most models we have used: even though they are trained on mostly unrelated data from the internet, we can leverage the pretrained weights for other tasks on a wide variety of texts. This is the core idea of transfer learning in NLP. Naturally, if the downstream task has similar textual structure as the pretraining texts the transfer works better, so if we can bring the pretraining task closer to the downstream objective we could potentially improve the transfer.

Let's think about this in terms of our concrete use case: BERT is pretrained on the BookCorpus and English Wikipedia, and texts containing code and GitHub issues are definitely a small niche in these datasets. If we pretrained BERT from scratch we could do it on a crawl of all of the issues on GitHub, for example. However, this would be expensive, and a lot of aspects about language that BERT has learned are still valid for GitHub issues. So is there a middle ground between retraining from scratch and just using the model as is for classification? There is, and it is called domain adaptation (which we also saw for question answering in Chapter 7). Instead of retraining the language model from scratch, we can continue training it on data from our domain. In this step we use the classic language model objective of predicting masked words, which means we don't need any labeled data. After that we can load the adapted model as a classifier and fine-tune it, thus leveraging the unlabeled data.

The beauty of domain adaptation is that compared to labeled data, unlabeled data is often abundantly available. Furthermore, the adapted model can be reused for many use cases. Imagine you want to build an email classifier and apply domain adaptation on all your historic emails. You can later use the same model for named entity recognition or another classification task like sentiment analysis, since the approach is agnostic to the downstream task.

Let's now see the steps we need to take to fine-tune a pretrained language model.

Fine-Tuning a Language Model

In this section we'll fine-tune the pretrained BERT model with masked language modeling on the unlabeled portion of our dataset. To do this we only need two new

concepts: an extra step when tokenizing the data and a special data collator. Let's start with the tokenization.

In addition to the ordinary tokens from the text the tokenizer also adds special tokens to the sequence, such as the `[CLS]` and `[SEP]` tokens that are used for classification and next sentence prediction. When we do masked language modeling, we want to make sure we don't train the model to also predict these tokens. For this reason we mask them from the loss, and we can get a mask when tokenizing by setting `return_special_tokens_mask=True`. Let's retokenize the text with that setting:

```
def tokenize(batch):
    return tokenizer(batch["text"], truncation=True,
                     max_length=128, return_special_tokens_mask=True)

ds_mlm = ds.map(tokenize, batched=True)
ds_mlm = ds_mlm.remove_columns(["labels", "text", "label_ids"])
```

What's missing to start with masked language modeling is the mechanism to mask tokens in the input sequence and have the target tokens in the outputs. One way we could approach this is by setting up a function that masks random tokens and creates labels for these sequences. But this would double the size of the dataset, since we would also store the target sequence in the dataset, and it would mean we would use the same masking of a sequence every epoch.

A much more elegant solution is to use a data collator. Remember that the data collator is the function that builds the bridge between the dataset and the model calls. A batch is sampled from the dataset, and the data collator prepares the elements in the batch to feed them to the model. In the simplest case we have encountered, it simply concatenates the tensors of each element into a single tensor. In our case we can use it to do the masking and label generation on the fly. That way we don't need to store the labels and we get new masks every time we sample. The data collator for this task is called `DataCollatorForLanguageModeling`. We initialize it with the model's tokenizer and the fraction of tokens we want to mask via the `mlm_probability` argument. We'll use this collator to mask 15% of the tokens, which follows the procedure in the BERT paper:

```
from transformers import DataCollatorForLanguageModeling, set_seed

data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer,
                                                mlm_probability=0.15)
```

Let's have a quick look at the data collator in action to see what it actually does. To quickly show the results in a `DataFrame`, we switch the return formats of the tokenizer and the data collator to NumPy:

```
set_seed(3)
data_collator.return_tensors = "np"
inputs = tokenizer("Transformers are awesome!", return_tensors="np")
```

```
outputs = data_collator([{"input_ids": inputs["input_ids"][0]}])

pd.DataFrame({
    "Original tokens": tokenizer.convert_ids_to_tokens(inputs["input_ids"][0]),
    "Masked tokens": tokenizer.convert_ids_to_tokens(outputs["input_ids"][0]),
    "Original input_ids": original_input_ids,
    "Masked input_ids": masked_input_ids,
    "Labels": outputs["labels"][0]}).T
```

	0	1	2	3	4	5
Original tokens	[CLS]	transformers	are	awesome	!	[SEP]
Masked tokens	[CLS]	transformers	are	awesome	[MASK]	[SEP]
Original input_ids	101	19081	2024	12476	999	102
Masked input_ids	101	19081	2024	12476	103	102
Labels	-100	-100	-100	-100	999	-100

We see that the token corresponding to the exclamation mark has been replaced with a mask token. In addition, the data collator returned a label array, which is –100 for the original tokens and the token ID for the masked tokens. As we have seen previously, the entries containing –100 are ignored when calculating the loss. Let's switch the format of the data collator back to PyTorch:

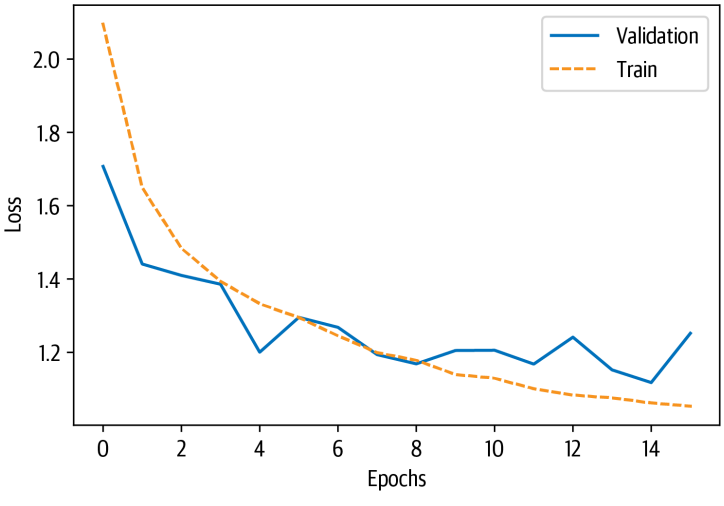
```
data_collator.return_tensors = "pt"
```

With the tokenizer and data collator in place, we are ready to fine-tune the masked language model. We set up the `TrainingArguments` and `Trainer` as usual:

```
from transformers import AutoModelForMaskedLM

training_args = TrainingArguments(
    output_dir = f"{model_ckpt}-issues-128", per_device_train_batch_size=32,
    logging_strategy="epoch", evaluation_strategy="epoch", save_strategy="no",
    num_train_epochs=16, push_to_hub=True, log_level="error", report_to="none")

trainer = Trainer(
        model=AutoModelForMaskedLM.from_pretrained("bert-base-uncased"),
        tokenizer=tokenizer, args=training_args, data_collator=data_collator,
        train_dataset=ds_mlm["unsup"], eval_dataset=ds_mlm["train"])

trainer.train()

trainer.push_to_hub("Training complete!")
```

We can access the trainer's log history to look at the training and validation losses of the model. All logs are stored in `trainer.state.log_history` as a list of dictionaries that we can easily load into a Pandas `DataFrame`. Since the training and validation loss are recorded at different steps, there are missing values in the dataframe. For this reason we drop the missing values before plotting the metrics:

```
df_log = pd.DataFrame(trainer.state.log_history)

(df_log.dropna(subset=["eval_loss"]).reset_index()["eval_loss"]
 .plot(label="Validation"))
df_log.dropna(subset=["loss"]).reset_index()["loss"].plot(label="Train")

plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(loc="upper right")
plt.show()
```



It seems that both the training and validation loss went down considerably. So let's check if we can also see an improvement when we fine-tune a classifier based on this model.

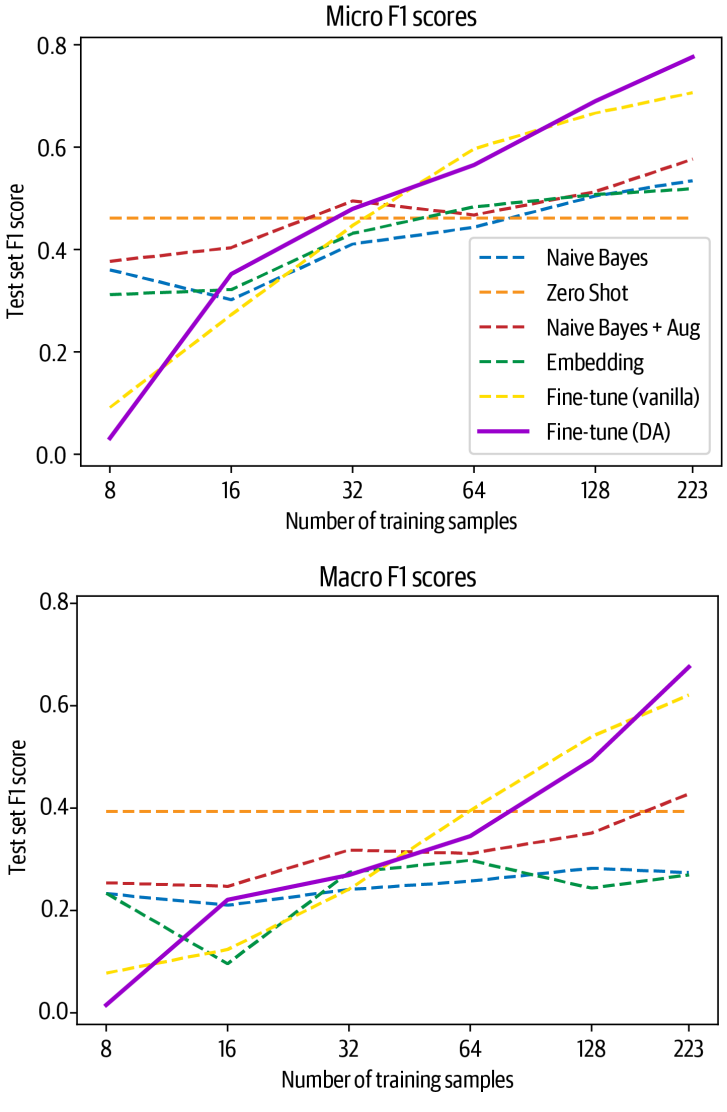
Fine-Tuning a Classifier

Now we'll repeat the fine-tuning procedure, but with the slight difference that we load our own custom checkpoint:

```
model_ckpt = f'{model_ckpt}-issues-128'
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels)
config.problem_type = "multi_label_classification"

for train_slice in train_slices:
    model = AutoModelForSequenceClassification.from_pretrained(model_ckpt,
                                                               config=config)
    trainer = Trainer(
        model=model,
        tokenizer=tokenizer,
        args=training_args_fine_tune,
        compute_metrics=compute_metrics,
        train_dataset=ds_enc["train"].select(train_slice),
        eval_dataset=ds_enc["valid"],
    )

    trainer.train()
    pred = trainer.predict(ds_enc['test'])
    metrics = compute_metrics(pred)
    # DA refers to domain adaptation
    macro_scores['Fine-tune (DA)'].append(metrics['macro f1'])
    micro_scores['Fine-tune (DA)'].append(metrics['micro f1'])
```

Comparing the results to the fine-tuning based on vanilla BERT, we see that we get an advantage especially in the low-data domain. We also gain a few percentage points in the regime where more labeled data is available:

```
plot_metrics(micro_scores, macro_scores, train_samples, "Fine-tune (DA)")
```



This highlights that domain adaptation can provide a slight boost to the model's performance with unlabeled data and little effort. Naturally, the more unlabeled data and the less labeled data you have, the more impact you will get with this method. Before we conclude this chapter, we'll show you a few more tricks for taking advantage of unlabeled data.

Advanced Methods

Fine-tuning the language model before tuning the classification head is a simple yet reliable method to boost performance. However, there are sophisticated methods than can leverage unlabeled data even further. We summarize a few of these methods here, which should provide a good starting point if you need more performance.

Unsupervised data augmentation

The key idea behind unsupervised data augmentation (UDA) is that a model's predictions should be consistent for an unlabeled example and a slightly distorted one. Such distortions are introduced with standard data augmentation strategies such as token replacement and back translation. Consistency is then enforced by minimizing the KL divergence between the predictions of the original and distorted examples. This process is illustrated in Figure 9-5, where the consistency requirement is incorporated by augmenting the cross-entropy loss with an additional term from the unlabeled examples. This means that one trains a model on the labeled data with the standard supervised approach, but constrains the model to make consistent predictions on the unlabeled data.

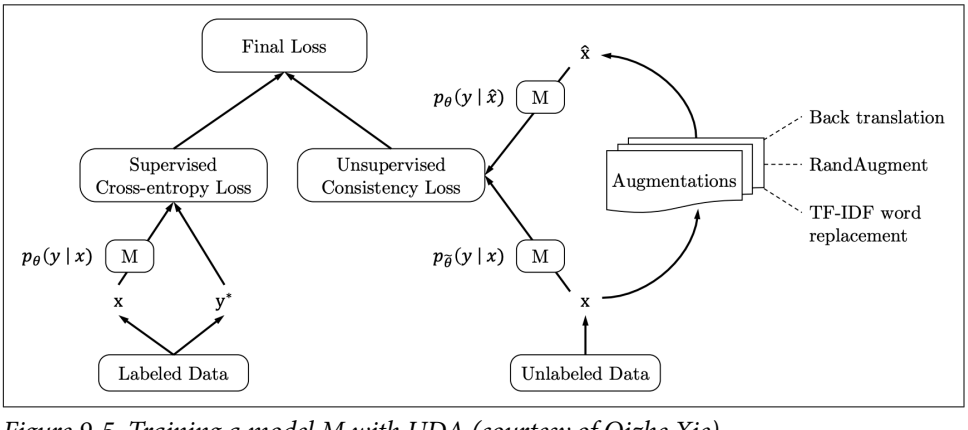


Figure 9-5. Training a model M with UDA (courtesy of Qizhe Xie)

The performance of this approach is quite impressive: with a handful of labeled examples, BERT models trained with UDA get similar performance to models trained on thousands of examples. The downside is that you need a data augmentation pipeline, and training takes much longer since you need multiple forward passes to generate the predicted distributions on the unlabeled and augmented examples.

Uncertainty-aware self-training

Another promising method to leverage unlabeled data is uncertainty-aware self-training (UST). The idea here is to train a teacher model on the labeled data and then

use that model to create pseudo-labels on the unlabeled data. Then a student is trained on the pseudo-labeled data, and after training it becomes the teacher for the next iteration.

One interesting aspect of this method is how the pseudo-labels are generated: to get an uncertainty measure of the model's predictions the same input is fed several times through the model with dropout turned on. Then the variance in the predictions gives a proxy for the certainty of the model on a specific sample. With that uncertainty measure the pseudo-labels are then sampled using a method called Bayesian Active Learning by Disagreement (BALD). The full training pipeline is illustrated in Figure 9-6.

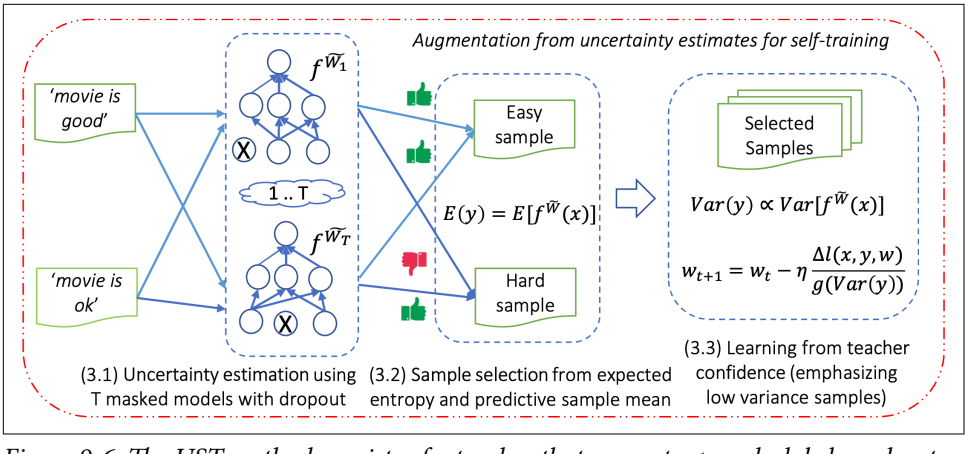


Figure 9-6. The UST method consists of a teacher that generates pseudo-labels and a student that is subsequently trained on those labels; after the student is trained it becomes the teacher and the step is repeated (courtesy of Subhabrata Mukherjee)[9]

With this iteration scheme the teacher continuously gets better at creating pseudo-labels, and thus the model's performance improves. In the end this approach gets within a few percent of models trained on the full training data with thousands of samples and even beats UDA on several datasets.

Now that we've seen a few advanced methods, let's take a step back and summarize what we've learned in this chapter.

9 S. Mukherjee and A.H. Awadallah, "Uncertainty-Aware Self-Training for Few-Shot Text Classification", (2020).

Conclusion

In this chapter we've seen that even if we have only a few or even no labels, not all hope is lost. We can utilize models that have been pretrained on other tasks, such as the BERT language model or GPT-2 trained on Python code, to make predictions on the new task of GitHub issue classification. Furthermore, we can use domain adaptation to get an additional boost when training the model with a normal classification head.

Which of the presented approaches will work best on a specific use case depends on a variety of aspects: how much labeled data you have, how noisy is it, how close the data is to the pretraining corpus, and so on. To find out what works best, it is a good idea to set up an evaluation pipeline and then iterate quickly. The flexible API of 🤗 Transformers allows you to quickly load a handful of models and compare them without the need for any code changes. There are over 10,000 models on the Hugging Face Hub, and chances are somebody has worked on a similar problem in the past and you can build on top of this.

One aspect that is beyond the scope of this book is the trade-off between a more complex approach like UDA or UST and getting more data. To evaluate your approach, it makes sense to at least build a validation and test set early on. At every step of the way you can also gather more labeled data. Usually annotating a few hundred examples is a matter of a couple of hours' or a few days' work, and there are many tools that can assist you in doing so. Depending on what you are trying to achieve, it can make sense to invest some time in creating a small, high-quality dataset rather than engineering a very complex method to compensate for the lack thereof. With the methods we've presented in this chapter you can ensure that you get the most value out of your precious labeled data.

Here, we have ventured into the low-data regime and seen that transformer models are still powerful even with just a hundred examples. In the next chapter we'll look at the complete opposite case: we'll see what we can do when we have hundreds of gigabytes of data and a lot of compute. We'll train a large transformer model from scratch to autocomplete code for us.