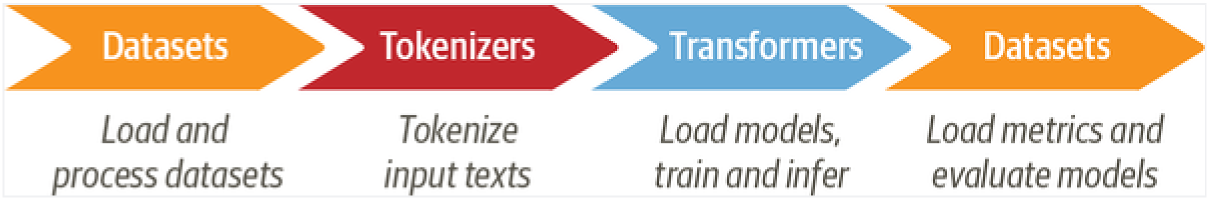
[Chapter2] Text Classification

1. Introduction

Text classification is one of the most common tasks in NLP; it can be used for a broad range of applications, such as tagging customer feedback into categories or routing support tickets according to their language.
In this chapter, we will be using DistilBERT, which achieves comparable performance to BERT, while being significantly smaller and more efficient.

1.1 Transformer Model Training Pipeline


Datasets
Load and process datasets
Tokenizers
Tokenize input texts
Transformers
Load models, train and infer
Datasets
Load metrics and evaluate models

1.2 Datasets

A short view of the datasets

```
from datasets import list_datasets

all_datasets = list_datasets()
print(f"There are {len(all_datasets)} datasets currently available on the Hub")
print(f"The first 10 are: {all_datasets[:10]}")
```

Output:

```
There are 1753 datasets currently available on the Hub
The first 10 are: ['acronym_identification', 'ade_corpus_v2', 'adversarial_qa',
'aeslc', 'afrikaans_ner_corpus', 'ag_news', 'ai2_arc', 'air_dialogue',
'ajgt_twitter_ar', 'allegro_reviews']
```

2. Load Data and Inspection

Dataset: CARER: Contextualized Affect Representations for Emotion Recognition

2.1 Load Dataset

2.1.1 From the Huggingface Hub

```
from datasets import load_dataset

emotions = load_dataset("emotion")
emotions
```

Output: (DatasetDict)

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 16000
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
})
```

2.1.2 Load External Dataset:

Common formats:

Data format	Loading script	Example
CSV	`csv`	`load_dataset("csv", data_files="my_file.csv")`
Text	`text`	`load_dataset("text", data_files="my_file.txt")`
JSON	`json`	`load_dataset("json", data_files="my_file.jsonl")`

Some other real-life examples:

From CSV online:

```
dataset_url = "https://www.dropbox.com/s/1pzkadrvffbqw6o/train.txt?dl=1"
emotions_remote = load_dataset("csv", data_files=dataset_url, sep=";",
                               names=["text", "label"])
```

Load from txt files:

```
wiki_dataset = load_dataset("text", data_files=["train/train/*.txt",
"valid/valid/*.txt"], split="train")
```

OR

```
wiki_dataset_with_test = load_dataset("text", data_files={"train": "train/train/*.txt",
"test": "valid/valid/*.txt"})
```

```
dataset = load_dataset('csv', data_files={'train': 'Resume_train.csv', 'test':
'Resume_test.csv' })
```

Extract Nepali Texts from OSCAR and cc100 Dataset:

```
original_ne = load_dataset('oscar', "unshuffled_original_ne", split='train',
streaming=False)
cc100_ne = load_dataset("cc100", split='train', lang="ne", streaming=False)
```

Splits and Slicing

```
# The full `train` split.
train_ds = datasets.load_dataset('bookcorpus', split='train')

# The full `train` split and the full `test` split as two distinct datasets.
train_ds, test_ds = datasets.load_dataset('bookcorpus', split=['train', 'test'])

# The full `train` and `test` splits, concatenated together.
train_test_ds = datasets.load_dataset('bookcorpus', split='train+test')

# From record 10 (included) to record 20 (excluded) of `train` split.
train_10_20_ds = datasets.load_dataset('bookcorpus', split='train[10:20]')

# The first 10% of `train` split.
train_10pct_ds = datasets.load_dataset('bookcorpus', split='train[:10%]')

# The first 10% of `train` + the last 80% of `train`.
train_10_80pct_ds = datasets.load_dataset('bookcorpus',
split='train[:10%]+train[-80%:]')
```

More Examples: [Splits and slicing]

2.2 Data Inspection

```
emotions = load_dataset("emotion")
train_ds = emotions["train"]
train_ds
```

Output: (Dataset)

```
Dataset({
    features: ['text', 'label'],
    num_rows: 16000
})
```

Other methods:

```
len(train_ds)
train_ds[0]
train_ds.column_names
train_ds.features
print(train_ds[:5])
print(train_ds["text"][:5])
```

Output of each:

```
16000

{'label': 0, 'text': 'i didnt feel humiliated'}

['text', 'label']

{'text': Value(dtype='string', id=None), 'label': ClassLabel(num_classes=6,
names=['sadness', 'joy', 'love', 'anger', 'fear', 'surprise'], names_file=None,
id=None)}

{'text': ['i didnt feel humiliated', 'i can go from feeling so hopeless to so
damned hopeful just from being around someone who cares and is awake', 'im
grabbing a minute to post i feel greedy wrong', 'i am ever feeling nostalgic
about the fireplace i will know that it is still on the property', 'i am feeling
grouchy'], 'label': [0, 0, 3, 2, 3]}

['i didnt feel humiliated', 'i can go from feeling so hopeless to so damned
hopeful just from being around someone who cares and is awake', 'im grabbing a
minute to post i feel greedy wrong', 'i am ever feeling nostalgic about the
fireplace i will know that it is still on the property', 'i am feeling grouchy']
```

2.2.1 Datasets to Pandas Dataframe

```
import pandas as pd

emotions.set_format(type="pandas")
df = emotions["train"][:]
df.head()
```

text	label
i didnt feel humiliated	0
i can go from feeling so hopeless to so damned...	0
im grabbing a minute to post i feel greedy wrong	3
i am ever feeling nostalgic about the fireplac...	2
i am feeling grouchy	3

2.2.2 Data Imbalance:

- Randomly oversample the minority class.
- Randomly undersample the majority class.
- Gather more labeled data from the underrepresented classes.”

Resources: Imbalanced-learn library

Note: Make sure that you don’t apply sampling methods before creating your train/test splits prevent data leakage

```
import matplotlib.pyplot as plt

df["label_name"].value_counts(ascending=True).plot.barh()
plt.title("Frequency of Classes")
plt.show()
```

2.2.3 Input Sequence Length:

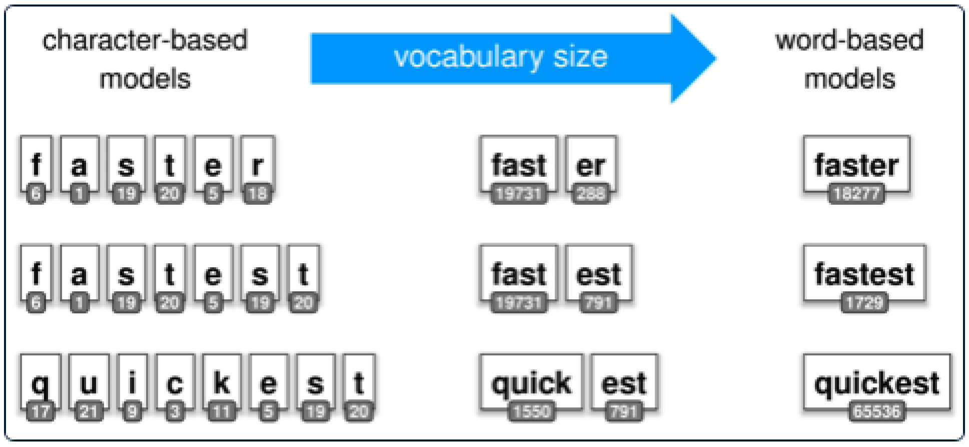
Why do we consider the length of inputs?

Transformer models have a maximum input sequence length that is referred to as the *maximum context size*. For applications using DistilBERT, the maximum context size is 512 tokens, which amounts to a few paragraphs of text.

```
df["Words Per Tweet"] = df["text"].str.split().apply(len)
df.boxplot("Words Per Tweet", by="label_name", grid=False,
           showfliers=False, color="black")
plt.suptitle("")
plt.xlabel("")
plt.show()
```

3. Tokenization

Tokenization is the step of breaking down a string into the atomic units used in the model. Transformer models like DistilBERT cannot receive raw strings as input; instead, they assume the text has been *tokenized* and *encoded* as numerical vectors.



From character-based to word-based tokenization.

When using pretrained models, it is *really* important to make sure that you use the same tokenizer that the model was trained with. From the model's perspective, switching the tokenizer is like shuffling the vocabulary.

3.1 Character Tokenization

```
# Tokenization
text = "Tokenizing text is a core task of NLP."
tokenized_text = list(text)

# Numeralication
token2idx = {ch: idx for idx, ch in enumerate(sorted(set(tokenized_text)))}
input_ids = [token2idx[token] for token in tokenized_text]
```

```
# Each token has now been mapped to a unique numerical identifier
input_ids = [token2idx[token] for token in tokenized_text]
```

token2idx:

```
{' ': 0, '.': 1, 'L': 2, 'N': 3, 'P': 4, 'T': 5, 'a': 6, 'c': 7, 'e': 8, 'f': 9,
'g': 10, 'i': 11, 'k': 12, 'n': 13, 'o': 14, 'r': 15, 's': 16, 't': 17, 'x': 18,
'z': 19}
```

input_ids:

```
[5, 14, 12, 8, 13, 11, 19, 11, 13, 10, 0, 17, 8, 18, 17, 0, 11, 16, 0, 6, 0, 7,
14, 15, 8, 0, 17, 6, 16, 12, 0, 14, 9, 0, 3, 2, 4, 1]
```

The problem with this approach is that it creates a fictitious ordering between the names (characters in this case), and neural networks are *really* good at learning these kinds of relationships. So instead, we can create a new column for each category and assign a 1 where the category is true, and a 0 otherwise.

We can create the one-hot encodings in PyTorch by converting `input_ids` to a tensor and applying the `one_hot()` function as follows:

```
import torch
import torch.nn.functional as F


input_ids = torch.tensor(input_ids)
one_hot_encodings = F.one_hot(input_ids,num_classes=len(token2idx))
one_hot_encodings.shape
```

Output:

```
torch.Size([38, 20])
```

For each of the 38 input tokens we now have a one-hot vector with 20 dimensions, since our vocabulary consists of 20 unique characters.

In Pandas, this can be implemented with the `get_dummies()` function as follows:

```
categorical_df = pd.DataFrame(
    {"Name": ["Bumblebee", "Optimus Prime", "Megatron"], "Label ID": [0,1,2]})
pd.get_dummies(categorical_df["Name"])
```

Bumblebee	Megatron	Optimus Prime
1	0	0
0	0	1
0	1	0

3.1.1 Issues of character tokenization

- The Character-level tokenization ignores any structure in the text and treats the whole string as a stream of characters.
- Although this helps deal with misspellings and rare words, the main drawback is that linguistic structures such as words need to be *learned* from the data.
- This requires significant compute, memory, and data.

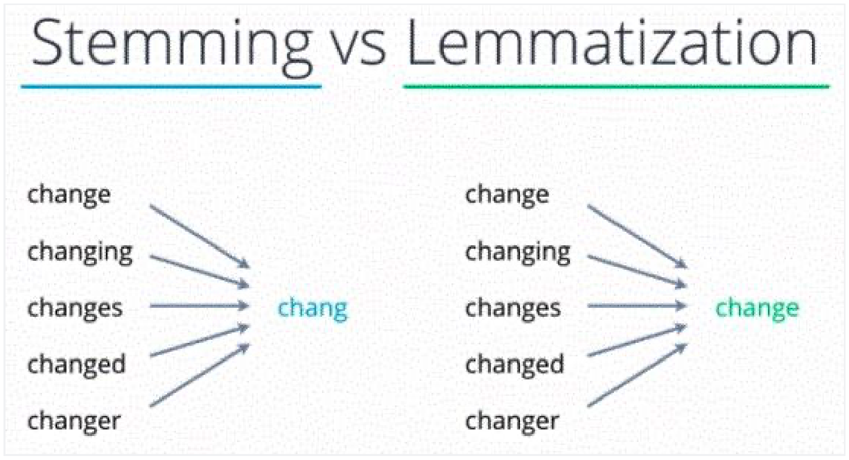
For this reason, character tokenization is rarely used in practice.

3.2 Word Tokenization

```
tokenized_text = text.split()
```

Issues:

- Punctuation is not accounted for, so `NLP.` is treated as a single token.
- Given that words can include declinations, conjugations, or misspellings, the size of the vocabulary can easily grow into the millions.
- Having a large vocabulary is a problem because it requires neural networks to have an enormous number of parameters.
- A common approach is to limit the vocabulary and discard rare words by considering, say, the 100,000 most common words in the corpus.
- Some word tokenizers have extra rules for punctuation. One can also apply stemming or lemmatization, which normalizes words to their stem (e.g., “great”, “greater”, and “greatest” all become “great”), at the expense of losing some information in the text.



3.3 Sub-word Tokenization

- The basic idea behind subword tokenization is to combine the best aspects of character and word tokenization.
- On the one hand, we want to split rare words into smaller units to allow the model to deal with complex words and misspellings.
- On the other hand, we want to keep frequent words as unique entities so that we can keep the length of our inputs to a manageable size.
- It is *learned* from the pretraining corpus using a mix of statistical rules and algorithms.

3.3.1 Wordpiece Tokenization (BERT and DistilBERT)

```
from transformers import AutoTokenizer

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

The `AutoTokenizer` class belongs to a larger set of “auto” classes whose job is to automatically retrieve the model’s configuration, pretrained weights, or vocabulary from the name of the checkpoint.

if you wish to load the specific class manually you can do so as well:

```
from transformers import DistilBertTokenizer

distilbert_tokenizer = DistilBertTokenizer.from_pretrained(model_ckpt)
```

Note: Default Location for the **model downloads** is `~/.cache/huggingface` You may change it as shown in the following example:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("Sakonii/de-berta-base-base-nepali",
use_auth_token=False, cache_dir="/mnt/storage0/common/UserName/FolderName/")
```

Tokenization steps and output

```
encoded_text = tokenizer(text)
tokens = tokenizer.convert_ids_to_tokens(encoded_text.input_ids)
tokenizer.convert_tokens_to_string(tokens)
tokenizer.vocab_size
tokenizer.model_max_length
```

Outputs:

```
{'input_ids': [101, 19204, 6026, 3793, 2003, 1037, 4563, 4708, 1997, 17953,
2361, 1012, 102], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}

['[CLS]', 'token', '##izing', 'text', 'is', 'a', 'core', 'task', 'of', 'nl',
'##p', '.', '[SEP]']

[CLS] tokenizing text is a core task of nlp. [SEP]

30522

512
```

- We can observe three things here. First, some special `[CLS]` and `[SEP]` tokens have been added to the start and end of the sequence. These tokens differ from model to model, but their main role is to indicate the start and end of a sequence.
- Second, the tokens have each been lowercased, which is a feature of this particular checkpoint.
- Finally, we can see that “tokenizing” and “NLP” have been split into two tokens, which makes sense since they are not common words. The `##` prefix in `##izing` and `##p` means that the preceding string is not whitespace; any token with this prefix should be merged with the previous token when you convert the tokens back to a string

IMP: Another interesting attribute to know about is the names of the fields that the model expects in its forward pass:

```
tokenizer.model_input_names
```

It lists the names of the fields that the model expects in its forward pass:

```
['input_ids', 'attention_mask']
```

3.3.2 BPE (Byte-Pair Encoding)

3.3.3 Unigram Tokenizers (Sentencepiece Tokenizer)

3.4 Tokenizing the Dataset

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

emotions_encoded = emotions.map(tokenize, batched=True, batch_size=None)
```

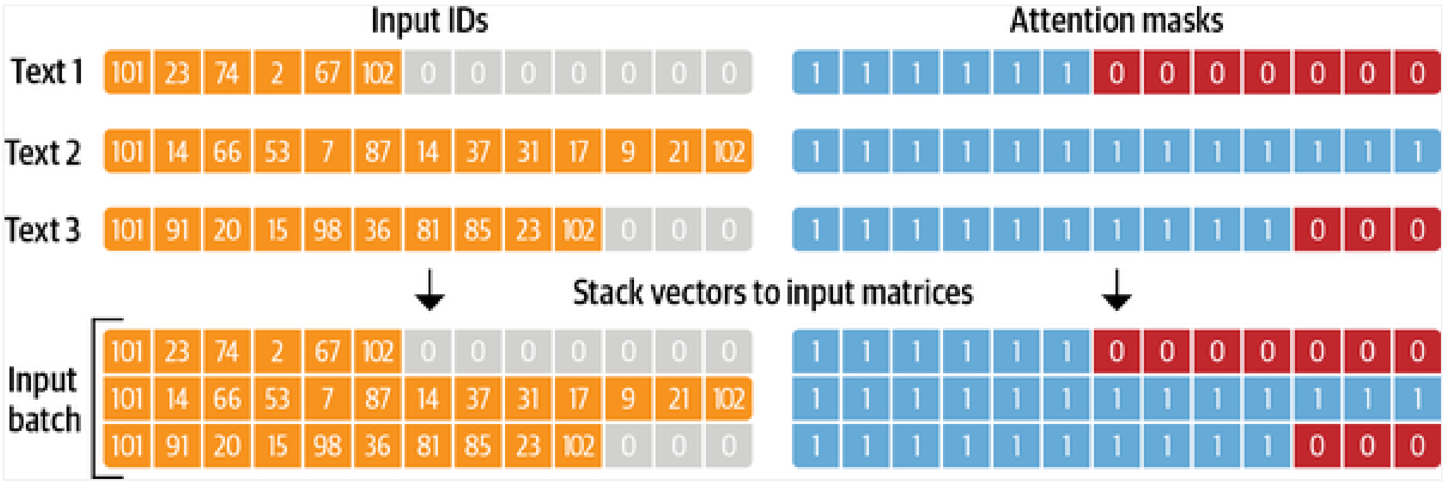
- `padding=True` will pad the examples with zeros to the size of the longest one in a batch.
- `truncation=True` will truncate the examples to the model’s maximum context size.

```
# Note: Code may not be correct
emotions_encoded["train"][:2]
```

Output:

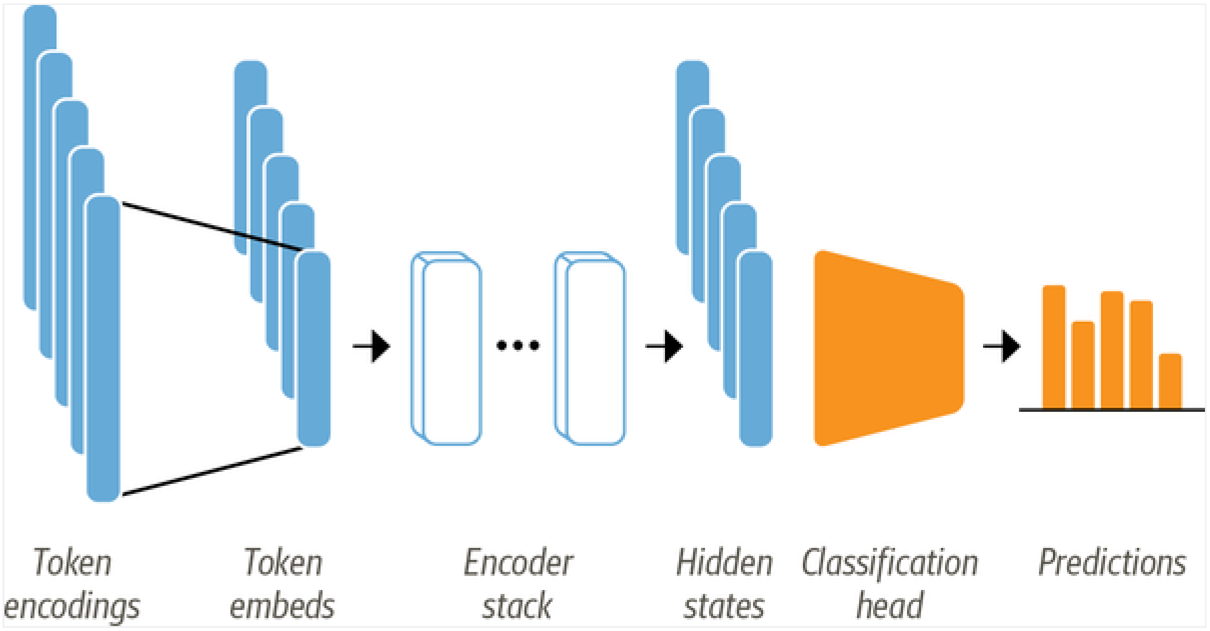
```
{'input_ids': [[101, 1045, 2134, 2102, 2514, 26608, 102, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0], [101, 1045, 2064, 2175, 2013, 3110, 2061, 20625, 2000,
2061, 9636, 17772, 2074, 2013, 2108, 2105, 2619, 2040, 14977, 1998, 2003, 8300,
102]], 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1]]}
```

- The first element of `input_ids` is shorter than the second, so zeros have been added to that element to make them the same length.
- These zeros have a corresponding `[PAD]` token in the vocabulary.



- In addition to returning the encoded tweets as input_ids, the tokenizer returns a list of attention_mask arrays.
- This is because we do not want the model to get confused by the additional padding tokens.
- The attention mask allows the model to ignore the padded parts of the input.

4. Training Text Classifier



- First, the text is tokenized and represented as one-hot vectors called *token encodings*. The size of the tokenizer vocabulary determines the dimension of the token encodings, and it usually consists of 20k–200k unique tokens.
- These token encodings are converted to *token embeddings*, which are vectors living in a lower-dimensional space.
- The token embeddings are then passed through the encoder block layers to yield a *hidden state* for each input token.

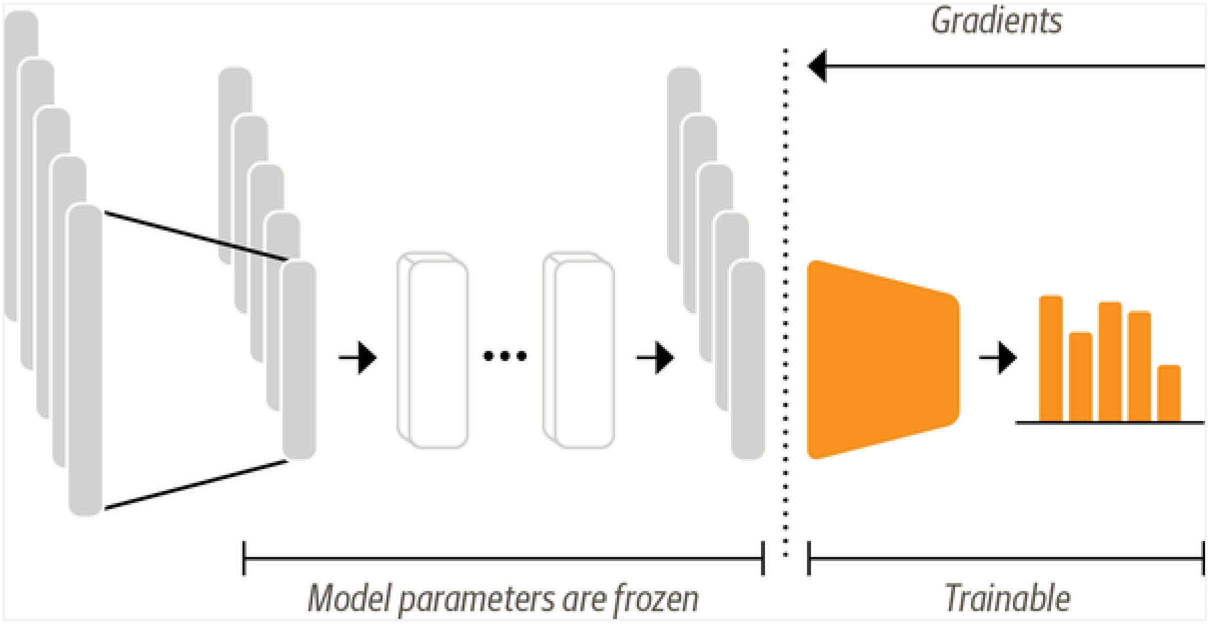
- For the pretraining objective of language modeling, each hidden state is fed to a layer that predicts the masked input tokens.
- For the classification task, we replace the language modeling layer with a classification layer.

Note: In practice, PyTorch skips the step of creating one-hot vectors for token encodings because multiplying a matrix with a one-hot vector is the same as selecting a column from the matrix. This can be done directly by getting the column with the token ID from the matrix.

We have two approaches for training Text Classifier:

- Feature Extractors
- Fine-tuning

4.1 Feature Extraction



- We freeze the body's weights during training and use the hidden states as features for the classifier.
- We can quickly train a small or shallow model.

4.1.1 Loading a model from the HUB:

```
from transformers import AutoModel

model_ckpt = "distilbert-base-uncased"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AutoModel.from_pretrained(model_ckpt).to(device)
```

- The `AutoModel` class converts the token encodings to embeddings, and then feeds them through the encoder stack to return the hidden states.

4.1.2 Extracting the last hidden states

```
text = "this is a test"
inputs = tokenizer(text, return_tensors="pt") # inputs['input_ids'].size() = [1, 6] i.e.
[batch_size, n_tokens]
inputs = {k:v.to(device) for k,v in inputs.items()}
with torch.no_grad():
    outputs = model(**inputs)
print(outputs)
```

```
BaseModelOutput(last_hidden_state=tensor([[[-0.1565, -0.1862,  0.0528,  ...,
-0.1188,  0.0662,  0.5470],
         [-0.3575, -0.6484, -0.0618,  ..., -0.3040,  0.3508,  0.5221],
         [-0.2772, -0.4459,  0.1818,  ..., -0.0948, -0.0076,  0.9958],
         [-0.2841, -0.3917,  0.3753,  ..., -0.2151, -0.1173,  1.0526],
         [ 0.2661, -0.5094, -0.3180,  ..., -0.4203,  0.0144, -0.2149],
         [ 0.9441,  0.0112, -0.4714,  ...,  0.1439, -0.7288, -0.1619]]],
       device='cuda:0'), hidden_states=None, attentions=None)
```

- Here we've used the `torch.no_grad()` context manager to disable the automatic calculation of the gradient. This is useful for inference since it reduces the memory footprint of the computations.
- The current model returns only one attribute, which is the last hidden state.

```
outputs.last_hidden_state.size() # torch.Size([1, 6, 768])
# Return vector for [CLS] token
outputs.last_hidden_state[:,0].size() # torch.Size([1, 768])
```

"Looking at the hidden state tensor, we see that it has the shape [batch_size, n_tokens, hidden_dim]. In other words, a 768-dimensional vector is returned for each of the 6 input tokens. For classification tasks, it is common practice to just use the hidden state associated with the [CLS] token as the input feature. Since this token appears at the start of each sequence, we can extract it by simply indexing into outputs.last_hidden_state.

Now we know how to get the last hidden state for a single string; let's do the same for the whole dataset:

```
def extract_hidden_states(batch):
    # Place model inputs on the GPU
    inputs = {k:v.to(device) for k,v in batch.items()
              if k in tokenizer.model_input_names}
    # Extract last hidden states
    with torch.no_grad():
        last_hidden_state = model(**inputs).last_hidden_state
    # Return vector for [CLS] token
    return {"hidden_state": last_hidden_state[:,0].cpu().numpy()}

# Since our model expects tensors as inputs
emotions_encoded.set_format("torch",
                            columns=["input_ids", "attention_mask", "label"])

emotions_hidden = emotions_encoded.map(extract_hidden_states, batched=True) # default
batch_size=1000
```

```
emotions_hidden["train"].column_names
```

```
['attention_mask', 'hidden_state', 'input_ids', 'label', 'text']
```

Now that we have the hidden states associated with each tweet, the next step is to train a classifier on them. To do that, we’ll need a feature matrix.

4.1.3 Creating a feature matrix

The preprocessed dataset now contains all the information we need to train a classifier on it. We will use the hidden states as input features and the labels as targets.

```
import numpy as np

X_train = np.array(emotions_hidden["train"]["hidden_state"])
X_valid = np.array(emotions_hidden["validation"]["hidden_state"])
y_train = np.array(emotions_hidden["train"]["label"])
y_valid = np.array(emotions_hidden["validation"]["label"])
X_train.shape, X_valid.shape
```

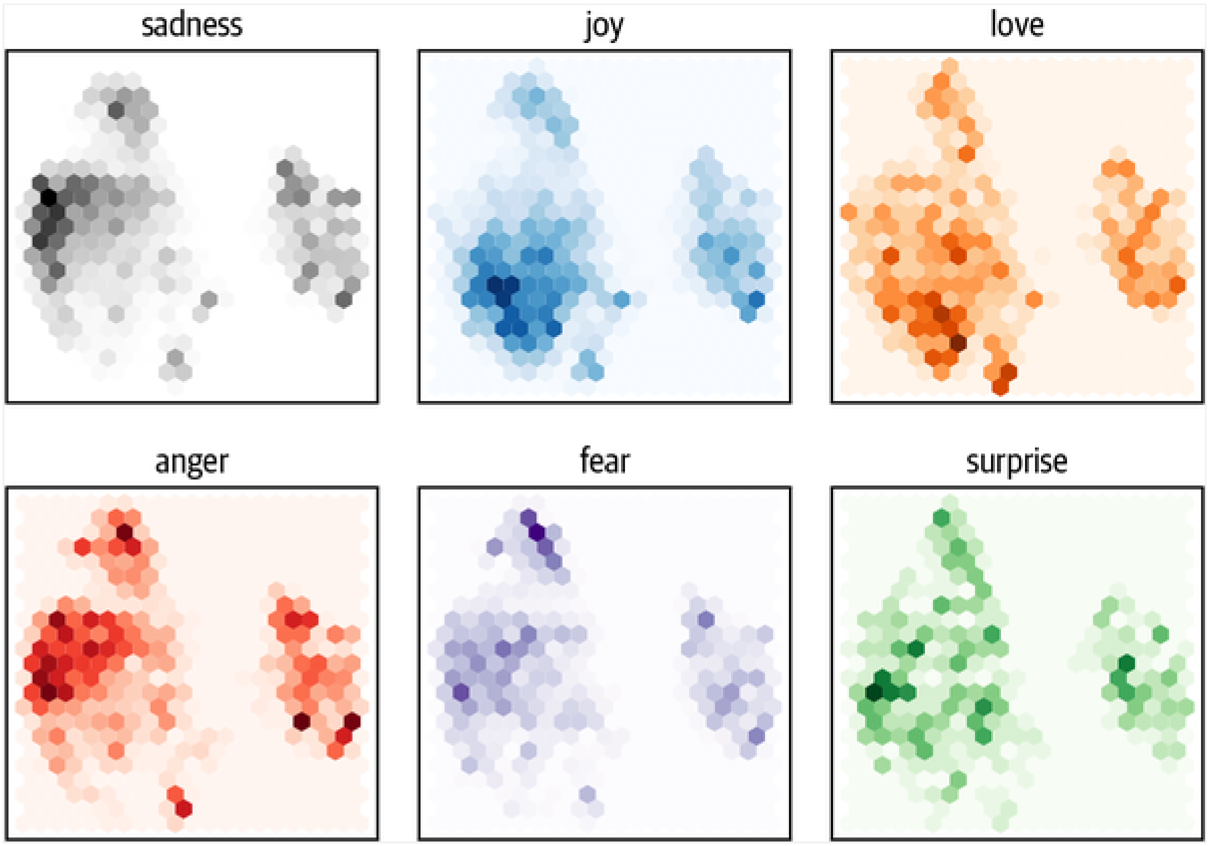
```
((16000, 768), (2000, 768))
```

4.1.4 Visualizing the training set

Since visualizing the hidden states in 768 dimensions is tricky to say the least, we'll use the powerful UMAP algorithm to project the vectors down to 2D.[7] Since UMAP works best when the features are scaled to lie in the [0,1] interval, we'll first apply a `MinMaxScaler` and then use the UMAP implementation from the `umap-learn` library to reduce the hidden states:

```
from umap import UMAP
from sklearn.preprocessing import MinMaxScaler

# Scale features to [0,1] range
X_scaled = MinMaxScaler().fit_transform(X_train)
# Initialize and fit UMAP
mapper = UMAP(n_components=2, metric="cosine").fit(X_scaled)
# Create a DataFrame of 2D embeddings
df_emb = pd.DataFrame(mapper.embedding_, columns=["X", "Y"])
df_emb["label"] = y_train
df_emb.head()
```

```
fig, axes = plt.subplots(2, 3, figsize=(7,5))
axes = axes.flatten()
cmaps = ["Greys", "Blues", "Oranges", "Reds", "Purples", "Greens"]
labels = emotions["train"].features["label"].names

for i, (label, cmap) in enumerate(zip(labels, cmaps)):
    df_emb_sub = df_emb.query(f"label == {i}")
    axes[i].hexbin(df_emb_sub["X"], df_emb_sub["Y"], cmap=cmap,
                   gridsize=20, linewidths=(0,))
    axes[i].set_title(label)
    axes[i].set_xticks([]), axes[i].set_yticks([])

plt.tight_layout()
plt.show()
```



- From this plot we can see some clear patterns: the negative feelings such as `sadness`, `anger`, and `fear` all occupy similar regions with slightly varying distributions. On the other hand, `joy` and `love` are well separated from the negative emotions and also share a similar space.

Note: These are only projections onto a lower-dimensional space. Just because some categories overlap does not mean that they are not separable in the original space. Conversely, if they are separable in the projected space they will be separable in the original space.

4.1.5 Training a simple classifier

Dummy Classifier:

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier(strategy="most_frequent")
dummy_clf.fit(X_train, y_train)
dummy_clf.score(X_valid, y_valid)
```

```
0.352
```

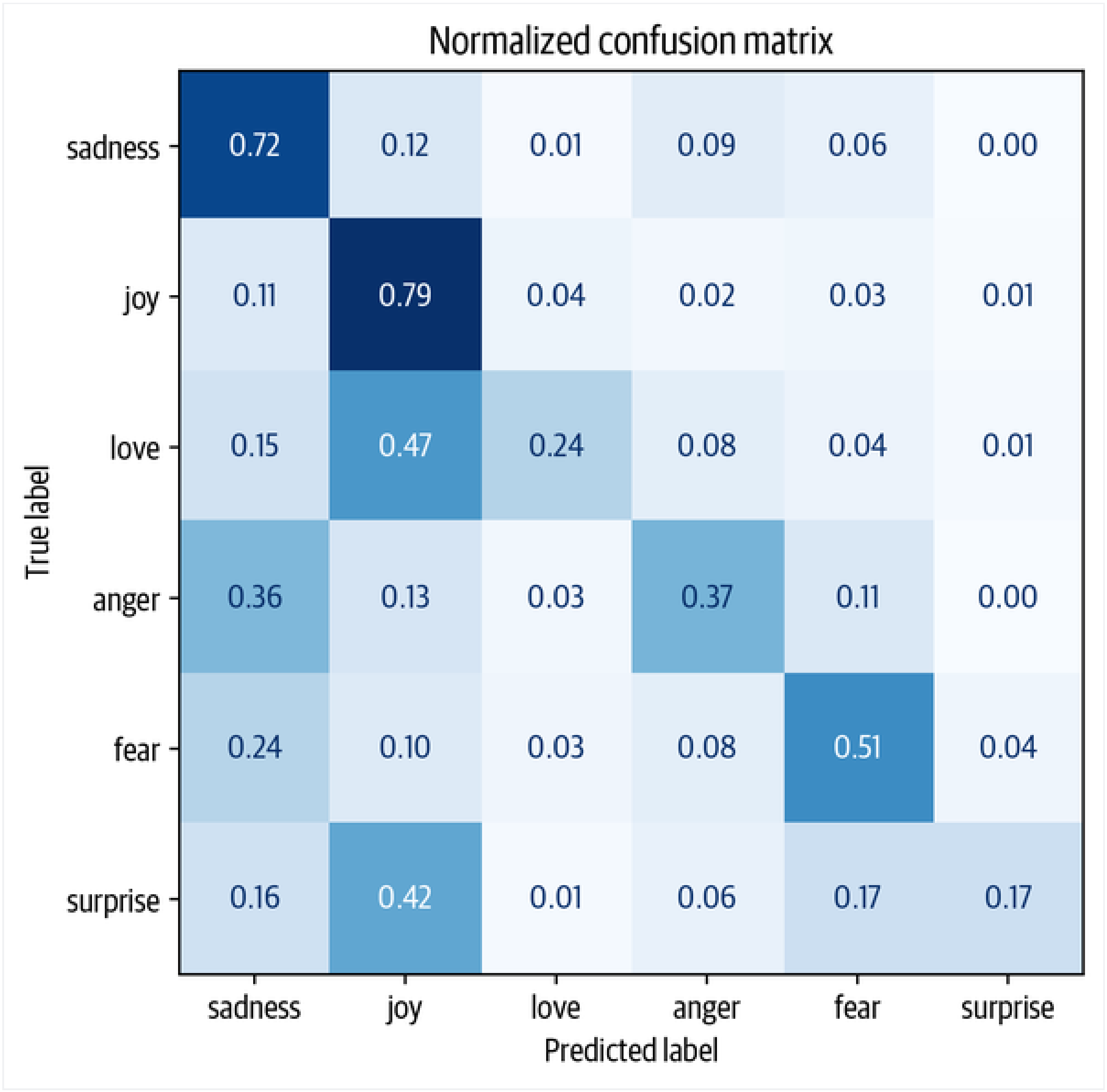
Logistic Regression:

```
from sklearn.linear_model import LogisticRegression

# We increase `max_iter` to guarantee convergence
lr_clf = LogisticRegression(max_iter=3000)
lr_clf.fit(X_train, y_train)
lr_clf.score(X_valid, y_valid)
```

```
0.633
```

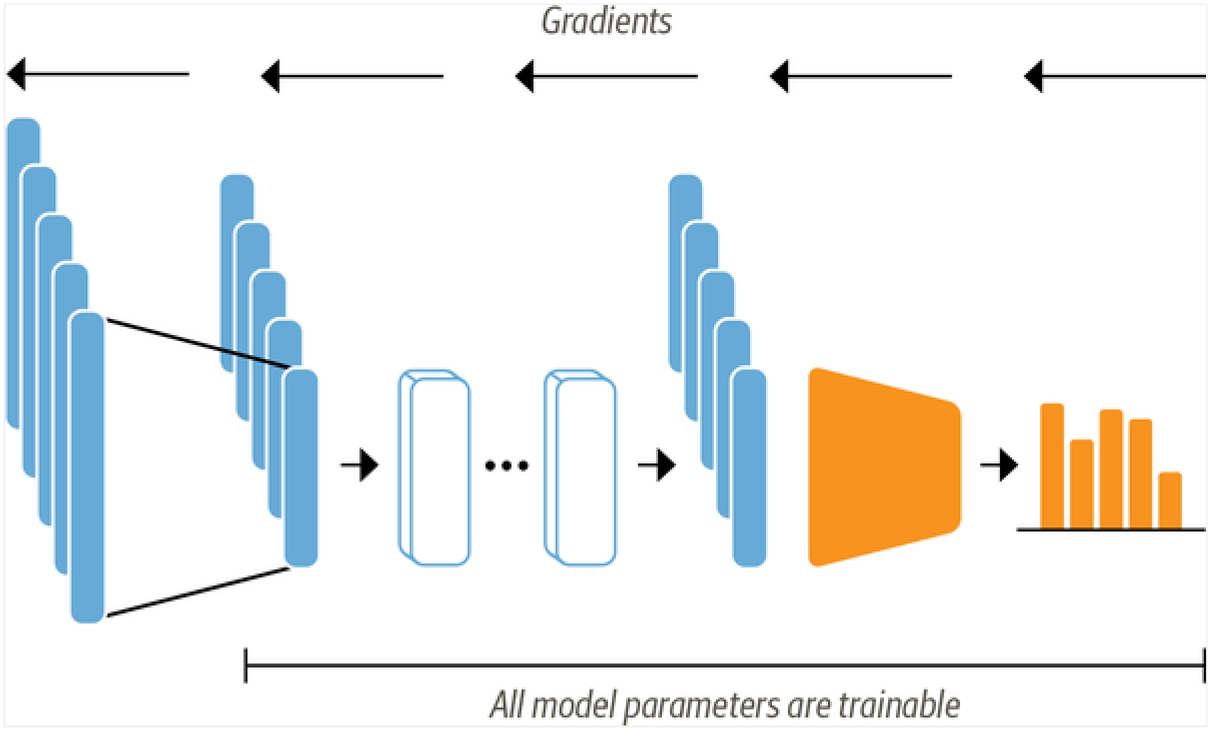
Plot Confusion Matrix:

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

def plot_confusion_matrix(y_preds, y_true, labels):
    cm = confusion_matrix(y_true, y_preds, normalize="true")
    fig, ax = plt.subplots(figsize=(6, 6))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
    disp.plot(cmap="Blues", values_format=".2f", ax=ax, colorbar=False)
    plt.title("Normalized confusion matrix")
    plt.show()

y_preds = lr_clf.predict(X_valid)
plot_confusion_matrix(y_preds, y_valid, labels)
```



- We can see that `anger` and `fear` are most often confused with `sadness`, which agrees with the observation we made when visualizing the embeddings.
- Also, `love` and `surprise` are frequently mistaken for `joy`.

4.2 Fine-tuning



4.2.1 Loading a pretrained model

AutoModel vs AutoModelForSequenceClassification (IMP): The only slight modification is that we use the `AutoModelForSequenceClassification` model instead of `AutoModel`. The difference is that the `AutoModelForSequenceClassification` model has a classification head on top of the pretrained model outputs, which can be easily trained with the base model.

```
from transformers import AutoModelForSequenceClassification

num_labels = 6
model = (AutoModelForSequenceClassification
         .from_pretrained(model_ckpt, num_labels=num_labels)
         .to(device))
```

4.2.2 Metrics

```
def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    f1 = f1_score(labels, preds, average="weighted")
    acc = accuracy_score(labels, preds)
    return {"accuracy": acc, "f1": f1}
```

4.2.3 Training

TrainingArguments: The most important argument to specify is `output_dir`, which is where all the artifacts from training are stored.

```
from huggingface_hub import notebook_login

notebook_login()

from transformers import Trainer, TrainingArguments

batch_size = 64
logging_steps = len(emotions_encoded["train"]) // batch_size
model_name = f"{model_ckpt}-finetuned-emotion"
training_args = TrainingArguments(output_dir=model_name,
                                  num_train_epochs=2,
                                  learning_rate=2e-5,
                                  per_device_train_batch_size=batch_size,
                                  per_device_eval_batch_size=batch_size,
                                  weight_decay=0.01,
                                  evaluation_strategy="epoch",
                                  disable_tqdm=False,
                                  logging_steps=logging_steps,
                                  push_to_hub=True,
                                  log_level="error")
```

Here we also set the batch size, learning rate, and number of epochs, and specify to load the best model at the end of the training run. With this final ingredient, we can instantiate and fine-tune our model with the `Trainer`:

```
from transformers import Trainer

trainer = Trainer(model=model, args=training_args,
                  compute_metrics=compute_metrics,
                  train_dataset=emotions_encoded["train"],
                  eval_dataset=emotions_encoded["validation"],
                  tokenizer=tokenizer)
trainer.train();
```

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.840900	0.327445	0.896500	0.892285
2	0.255000	0.220472	0.922500	0.922550

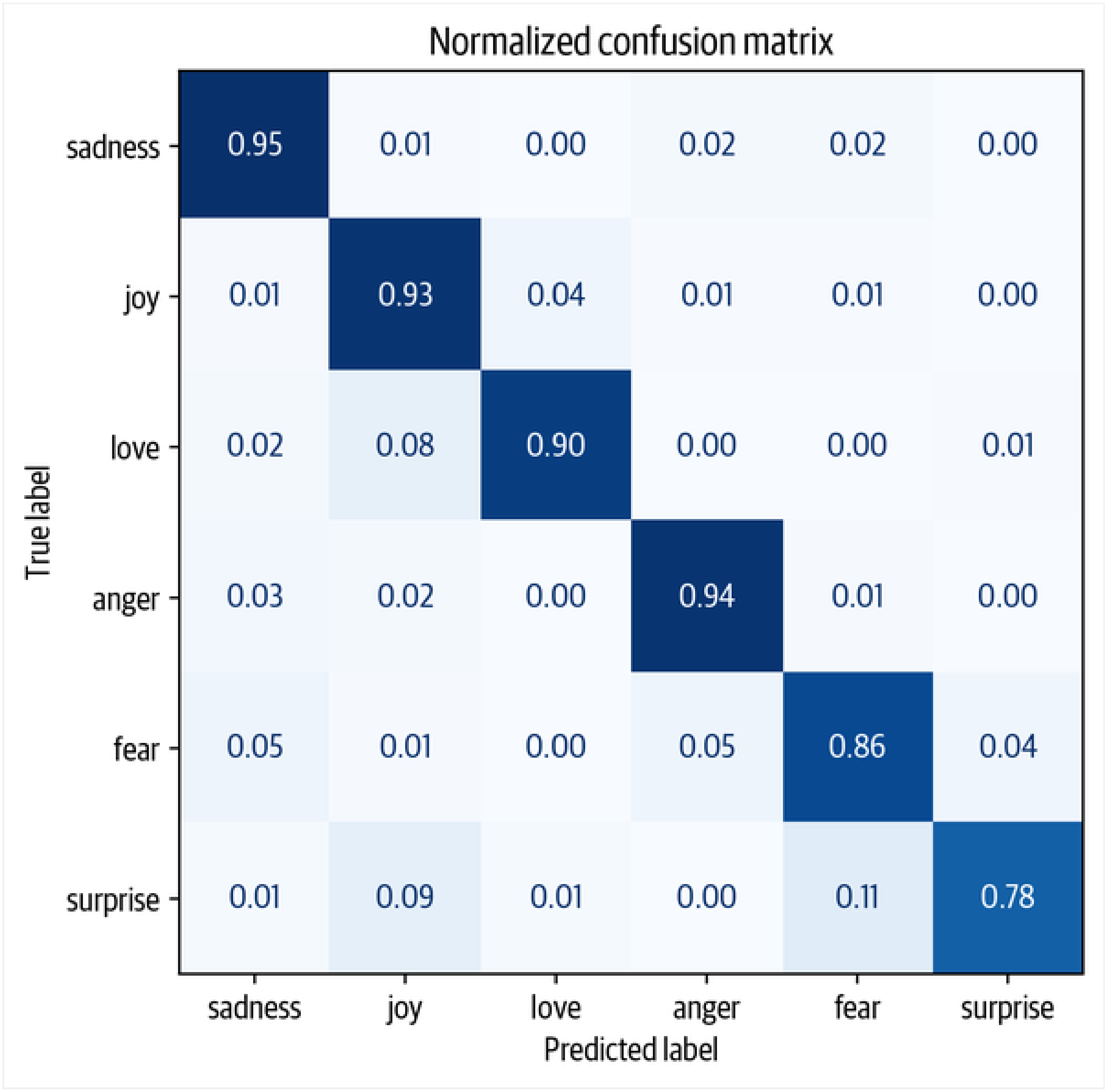
4.2.4 Evaluation

```
preds_output = trainer.predict(emotions_encoded["validation"])
preds_output.metrics
```

```
{'test_loss': 0.22047173976898193,
 'test_accuracy': 0.9225,
 'test_f1': 0.9225500751072866,
 'test_runtime': 1.6357,
 'test_samples_per_second': 1222.725,
 'test_steps_per_second': 19.564}
```

It also contains the raw predictions for each class. We can decode the predictions greedily using `np.argmax()`. This yields the predicted labels and has the same format as the labels returned by the Scikit-learn models in the feature-based approach:

```
y_preds = np.argmax(preds_output.predictions, axis=1)
plot_confusion_matrix(y_preds, y_valid, labels)
```



4.2.5 Error Analysis

A simple yet powerful technique is to sort the validation samples by the model loss. When we pass the label during the forward pass, the loss is automatically calculated and returned.

Let’s first have a look at the data samples with the highest losses

```
from torch.nn.functional import cross_entropy

def forward_pass_with_label(batch):
    # Place all input tensors on the same device as the model
    inputs = {k:v.to(device) for k,v in batch.items()
              if k in tokenizer.model_input_names}

    with torch.no_grad():
        output = model(**inputs)
        pred_label = torch.argmax(output.logits, axis=-1)
        loss = cross_entropy(output.logits, batch["label"].to(device),
                             reduction="none")
    # Place outputs on CPU for compatibility with other dataset columns
    return {"loss": loss.cpu().numpy(),
            "predicted_label": pred_label.cpu().numpy()}

# Convert our dataset back to PyTorch tensors
emotions_encoded.set_format("torch",
                            columns=["input_ids", "attention_mask", "label"])
# Compute loss values
emotions_encoded["validation"] = emotions_encoded["validation"].map(
    forward_pass_with_label, batched=True, batch_size=16)

emotions_encoded.set_format("pandas")
cols = ["text", "label", "predicted_label", "loss"]
df_test = emotions_encoded["validation"][:][cols]
df_test["label"] = df_test["label"].apply(label_int2str)
df_test["predicted_label"] = (df_test["predicted_label"]
                              .apply(label_int2str))

df_test.sort_values("loss", ascending=False).head(4)
```

We can now easily sort emotions_encoded by the losses in either ascending or descending order. The goal of this exercise is to detect one of the following:

- Wrong Labels
- Quirks of the dataset

text	label	predicted_label	loss
i feel that he was being overshadowed by the supporting characters	love	sadness	5.704531
i called myself pro life and voted for perry without knowing this information i would feel betrayed but moreover i would feel that i had	joy	sadness	5.484461

betrayed god by supporting a man who mandated a barely year old vaccine for little girls putting them in danger to financially support people close to him			
i guess i feel betrayed because i admired him so much and for someone to do this to his wife and kids just goes beyond the pale	joy	sadness	5.434768
i feel badly about reneging on my commitment to bring donuts to the faithful at holy family catholic church in columbus ohio	love	sadness	5.257482

Deep learning models are exceptionally good at finding and exploiting shortcuts to get to a prediction. For this reason, it is also worth investing time into looking at the examples that the model is most confident about, so that we can be confident that the model does not improperly exploit certain features of the text.

```
df_test.sort_values("loss", ascending=True).head(3)
```

text	label	predicted_label	loss
i feel try to tell me im ungrateful tell me im basically the worst daughter sister in the world	sadness	sadness	0.017331

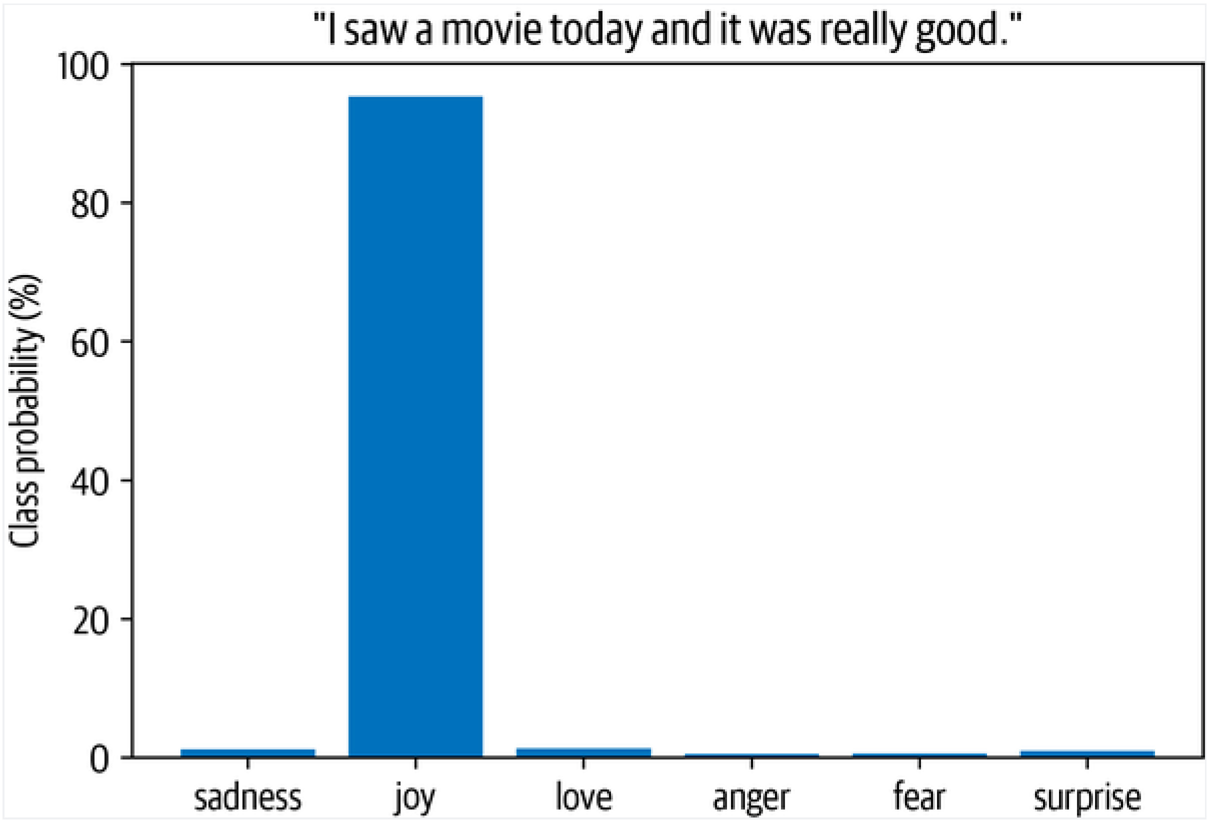
im kinda relieve but at the same time i feel disheartened	sadness	sadness	0.017392
i and feel quite ungrateful for it but i m looking forward to summer and warmth and light nights	sadness	sadness	0.017400

4.2.6 Saving / Sharing the Model

```
trainer.push_to_hub(commit_message="Training completed!")
```

4.2.7 Inference

```
from transformers import pipeline

# Change `transformersbook` to your Hub username
model_id = "transformersbook/distilbert-base-uncased-finetuned-emotion"
classifier = pipeline("text-classification", model=model_id)

custom_tweet = "I saw a movie today and it was really good."
preds = classifier(custom_tweet, return_all_scores=True)

preds_df = pd.DataFrame(preds[0])
plt.bar(labels, 100 * preds_df["score"], color='C0')
plt.title(f'"{custom_tweet}"')
plt.ylabel("Class probability (%)")
plt.show()
```


"I saw a movie today and it was really good."
Class probability (%)
0
20
40
60
80
100
sadness
joy
love
anger
fear
surprise