CARTE-Enbridge Bootcamp

Lab 2-2

Transforming customer service using AI

In this lab, we will use HuggingFace's transformers library to build a customer service chatbot. We will use the DistilBERT model to build a chatbot that can answer questions about the customer service policy of a company.

For this notebook to run in a reasonable time, it's essential that you enable GPU acceleration. To do this, go to Runtime > Change runtime type, and select GPU as the hardware accelerator.

In [1]:
```
# Check if GPU is enabled
import torch
if torch.cuda.is_available():
    device = torch.device("cuda")
    print(f'There are {torch.cuda.device_count()} GPU(s) available.')
elif torch.backends.mps.is_available():
    device = torch.device("mps")
    print(f'Running on a Mac with Metal Performance Shaders (MPS).')
else:
    device = torch.device("cpu")
    print('No GPU available, using the CPU instead (NOT RECOMMENDED!)')
```

```
There are 1 GPU(s) available.
```

First, we will import the necessary libraries and modules. If we are running this notebook on Google Colab, we will install the libraries that we need in the cell below. We are installing directly from GitHub, to ensure that we have the latest version of each library.

In [2]:
```
# Check if we are running on Google Colab
import sys
IN_COLAB = 'google.colab' in sys.modules
if IN_COLAB:
    !pip install -q -U transformers datasets evaluate accelerate
```

In [3]:
```
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from datasets import load_dataset
```

Load the dataset

Using the load_dataset() function from the datasets library, we will load the customer service dataset. We will be using a dataset of customer service queries that is designed for use in training chatbots.

The parts of the dataset that we are interested in are the customer's query and the 'intent' of the customer's query. The intent is the purpose of the customer's query. For example, if the customer's query is "What is your return policy?", the intent is "return policy". This makes it easier for the interaction to be routed to the correct department, or for collecting metrics on the types of queries that customers have.

In [4]:
```
data = load_dataset('bitext/Bitext-customer-support-llm-chatbot-training-dat
```

Let's take a quick look at a few random samples from the dataset. We will print the customer's query, the intent, and the response from the chatbot.

In [5]:
```
from random import choice
for i in range(5):
    sample = choice(data['train'])
    print(f'Question: {sample["instruction"]}')
    print(f'Customer Intent: {sample["intent"]}')
    print('---'*10)
```

```
Question: I do not know how I could cancel a {{Account Category}} account
Customer Intent: delete_account
------------------------------
Question: I need aasistance trying to update the address
Customer Intent: change_shipping_address
------------------------------
Question: I need help cancelling my subscription to your newsletter
Customer Intent: newsletter_subscription
------------------------------
Question: I don't know what to do to talk to a live agent
Customer Intent: contact_human_agent
------------------------------
Question: know if there is anything wrong with the refund
Customer Intent: track_refund
------------------------------
```

Preprocess the data

Now that we have our data downloaded, the next step is to preprocess it. For any model, each word in the model's vocabulary is assigned a unique number. The model doesn't see the actual words in the text, it only sees the numbers. So, we need to convert the text into numbers. This is called tokenization. Because we are fine-tuning an existing model, we can simply load the tokenizer that was created for that model. The tokenizer will convert the text into numbers, and also add any special tokens that the model needs. For example, the DistilBERT model needs a [CLS] token at the beginning of the text, and a [SEP] token at the end of the text. The tokenizer will add these tokens for us.

In [6]:
```
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
```

Let's see what happens when we tokenize a sample text:

In [7]:
```
question = "What is your return policy?"
tokens = tokenizer.encode(question)
print(tokens)
```

```
[101, 2054, 2003, 2115, 2709, 3343, 1029, 102]
```

Each of these values corresponds to one of the words in the text. We can use the tokenizer to convert these numbers back into words, and see what the tokenizer did to the text.

In [8]:
```
# Convert the tokens back into words
print(tokenizer.convert_ids_to_tokens(tokens))
```

```
['[CLS]', 'what', 'is', 'your', 'return', 'policy', '?', '[SEP]']
```

As we can see, the tokenizer converted the text into numbers, and added the [CLS] and [SEP] tokens.

Your Turn

Try tokenizing a few other sample texts.

- What happens if you put in a nonsense word?
- What happens if you put in a text that's longer than the maximum length of the model (512 tokens)?
- What happens if you put in an emoji, or another character that's not in the model's vocabulary?

In [9]:
```
def encode_and_decode(question):
    tokens = tokenizer.encode(question)
    recons = tokenizer.convert_ids_to_tokens(tokens)
    for token, id in zip(recons, tokens):
        print(f'{token:<12} {id}')
    print()
```

In [10]:
```
# Nonsense word
encode_and_decode("supercalifragilisticexpialidocious")

# Emoji
encode_and_decode("What is your return policy? 😊")
```

```
[CLS]        101
super        3565
##cal        9289
##if         10128
##rag        29181
##ilis       24411
##tic        4588
##ex         10288
##pia        19312
##lid        21273
##oc         10085
##ious       6313
[SEP]        102

[CLS]        101
what         2054
is           2003
your         2115
return       2709
policy       3343
?            1029
[UNK]        100
[SEP]        102
```

We are ready to tokenize our dataset. We will use the map() function to apply the tokenizer to each sample in the dataset. We will also set the maximum length of the input to 512 tokens.

There are a couple of parameters that we need to set. The first is padding. Although our input sentences can be any length, the model requires input to be a consistent size. The tokenizer can 'pad' the input by adding special tokens to the end of the input. The model will ignore these tokens, but they will allow the input to be the same size.

The second parameter is truncation. If the input is longer than the maximum length, the tokenizer will truncate the input. We will set both of these parameters to True.

In [11]:
```
def tokenize(batch):
    return tokenizer(batch['instruction'], padding=True, truncation=True)

tokenized_dataset = data.map(tokenize, batched=True, batch_size=len(data), r
```

The intents that we are trying to predict are text labels, like `payment_issue`, but the model is going to predict numbers. We can assign a number to each label, and then create a mapping from the label to the number. We will use this mapping to convert the labels into numbers.

In [12]:
```
intents = set(data['train']['intent']) # Get all of the unique intents
intents = list(intents) # Convert to a list, so that the order is consistent
```

```
intents.sort() # Sort alphabetically

id2label = {i: label for i, label in enumerate(intents)} # Create a mapping
label2id = {label: i for i, label in enumerate(intents)} # Create a mapping

print(id2label)
```

```
{0: 'cancel_order', 1: 'change_order', 2: 'change_shipping_address', 3: 'che
ck_cancellation_fee', 4: 'check_invoice', 5: 'check_payment_methods', 6: 'ch
eck_refund_policy', 7: 'complaint', 8: 'contact_customer_service', 9: 'conta
ct_human_agent', 10: 'create_account', 11: 'delete_account', 12: 'delivery_o
ptions', 13: 'delivery_period', 14: 'edit_account', 15: 'get_invoice', 16:
'get_refund', 17: 'newsletter_subscription', 18: 'payment_issue', 19: 'place
_order', 20: 'recover_password', 21: 'registration_problems', 22: 'review',
23: 'set_up_shipping_address', 24: 'switch_account', 25: 'track_order', 26:
'track_refund'}
```

In [13]:
```
ids = [label2id[label] for label in data['train']['intent']] # Get a list of
tokenized_dataset = tokenized_dataset['train'].add_column('label', ids) # Ad
```

```
/home/alex/mambaforge/envs/enbridge_pytorch/lib/python3.11/site-packages/dat
asets/table.py:1387: FutureWarning: promote has been superseded by mode='def
ault'.
  return cls._concat_blocks(pa_tables_to_concat_vertically, axis=0)
```

We are almost finished preparing our data! The last thing we need to do is split the dataset into a training set and a test set. We will use 80% of the data for training, and 20% for testing.

In [14]:
```
tokenized_dataset = tokenized_dataset.train_test_split(test_size=0.2)
```

We can confirm that the data looks how we expect by printing out the object:

In [15]:
```
tokenized_dataset
```

Out[15]:
```
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'attention_mask', 'label'],
        num_rows: 21497
    })
    test: Dataset({
        features: ['input_ids', 'attention_mask', 'label'],
        num_rows: 5375
    })
})
```

We also load a "data collator". This is a function that combines the data into batches, to make it easier for the model to process. We will use the default data collator, which simply combines the data into batches, and adds padding to the end of the input.

In [16]:
```
from transformers import DataCollatorWithPadding
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Fine-tune the model

Now that we have our data ready, we can fine-tune the model. As with the earlier stages, HuggingFace makes it very easy to load an existing model. Because the task we are trying to solve is text classification, we will use the AutoModelForSequenceClassification class. This class will load a model that has already been trained on a text classification task, and fine-tune it on our dataset. We will use the DistilBERT model, which is a smaller version of the BERT model. This will allow us to train the model faster, and use less memory.

There are a few arguments that we have to pass:

- The model name. We will use the DistilBERT model, which is a smaller version of the BERT model.
- The number of labels (or possible intents) in our dataset. This is the number of outputs that the model will have. We can get this number from the length of the intents list.
- The mapping from the label to the index. This is the mapping that we created earlier, which converts the label to a number.
- The mapping from the index to the label. This is the mapping that we created earlier, which converts the number back into a label.

You might get a warning that some of the weights are not initialized. This is fine, because we are fine-tuning the model, so these weights will be updated during training.

In [17]:
```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-
```

```
Some weights of DistilBertForSequenceClassification were not initialized fro
m the model checkpoint at distilbert-base-uncased and are newly initialized:
['classifier.weight', 'classifier.bias', 'pre_classifier.bias', 'pre_classif
ier.weight']
You should probably TRAIN this model on a down-stream task to be able to use
it for predictions and inference.
```

Next, we define the arguments for training the model. Each argument is explained in the code below.

In [18]:
```
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=3,              # total number of training epochs
    per_device_train_batch_size=16,  # batch size per device during training
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
    logging_steps=10,                # How often to log results
    evaluation_strategy='epoch',     # How often to evaluate the model - onc
```

```
    save_strategy='epoch',           # How often to save the model - once pe
    load_best_model_at_end=True,     # At the end of training, load the mode
)

# Move everything to the GPU/MPS, if available
model.to(device)

trainer = Trainer(
    model=model,                                # the model to be trained
    args=training_args,                         # training arguments, defin
    train_dataset=tokenized_dataset['train'],   # training dataset
    eval_dataset=tokenized_dataset['test'],     # evaluation dataset
    data_collator=data_collator,                # data collator
)
```

Now we can train the model! This will take around 5 minutes on Google Colab.

In [19]:
```
trainer.train()
```

```
You're using a DistilBertTokenizerFast tokenizer. Please note that with a fa
st tokenizer, using the `__call__` method is faster than using a method to e
ncode the text followed by a call to the `pad` method to get a padded encodi
ng.
```

[4032/4032 02:49, Epoch 3/3]

Epoch	Training Loss	Validation Loss
1	0.002600	0.013265
2	0.000400	0.008115
3	0.000300	0.008019

Out[19]:
```
TrainOutput(global_step=4032, training_loss=0.0815903501316493, metrics={'t
rain_runtime': 169.7203, 'train_samples_per_second': 379.984, 'train_steps_
per_second': 23.757, 'total_flos': 335857764228948.0, 'train_loss': 0.08159
03501316493, 'epoch': 3.0})
```

In [20]:
```
# You can save your model to disk
trainer.save_model("./models")
```

In [21]:
```
# ...and reload it at any time in the future!
model = AutoModelForSequenceClassification.from_pretrained("./models")
```

In [22]:
```
from transformers import pipeline
classifier = pipeline('text-classification', model=model, tokenizer=tokenize
```

In [23]:
```
# Let's try it out!
classifier("What is your return policy?")
```

Out[23]:
```
[{'label': 'check_refund_policy', 'score': 0.9888862371444702}]
```

Your Turn

Try asking a few questions to the chatbot.

- What happens if you ask a question that is not in the dataset?
- What happens if you ask a question that is in the dataset, but is phrased differently?
- What happens if you ask a question that has a typo? Or a word that is not in the model's vocabulary?
- What happens if you ask a question that is not in English?

In [24]:
```
# Question not in the dataset
print(classifier("What is your favorite color?"))

# Question in the dataset, but phrased differently
print(classifier("Can you tell me about your return policy?"))

# Question with a typo
print(classifier("What is your return polcy?"))

# Question with a word that is not in the model's vocabulary
print(classifier("What is your return policy? 😊"))

# Question not in English
print(classifier("¿Cuál es su política de devolución?"))
```

```
[{'label': 'place_order', 'score': 0.5761329531669617}]
[{'label': 'check_refund_policy', 'score': 0.993984043598175}]
[{'label': 'set_up_shipping_address', 'score': 0.44598865509033203}]
[{'label': 'check_refund_policy', 'score': 0.9906986355781555}]
[{'label': 'check_refund_policy', 'score': 0.8541324138641357}]
```

Model Evaluation

Now that we have trained our model, we can evaluate it. We will get the predictions for the test set, and calculate the accuracy.

In [25]:
```
from sklearn.metrics import accuracy_score

predictions = trainer.predict(tokenized_dataset['test'])
predictions = predictions.predictions.argmax(-1)
labels = tokenized_dataset['test']['label']

accuracy = accuracy_score(labels, predictions)
print(f'Accuracy: {accuracy:.2f}')
```

```
Accuracy: 1.00
```

Wow! As we can see, the model is very accurate. This is because we are using a model that has already been trained on a similar task. If we were training a model from scratch, we would expect the accuracy to be much lower.

Your Turn

When evaluation performance of a new model, it's often helpful to look at the examples that the model gets wrong. This can help you understand what types of errors the model is making, and how you might improve the model.

In the cell below, print out the examples that the model gets wrong. You can use the `predictions` and `labels` variables from the previous cell.

- What types of errors is the model making?
- Are there any patterns in the errors that the model is making?

In [26]:
```
import pandas as pd
df = []
for i, (k,v) in enumerate(zip(predictions, labels)):
    if k != v:
        df.append({'prediction': id2label[k], 'label': id2label[v], 'text':
df = pd.DataFrame(df)
df
```

Out[26]:

	prediction	label	text
0	registration_problems	payment_issue	[[CLS], i, get, an, error, when, i, try, to, c...
1	cancel_order	change_order	[[CLS], help, to, moi, ##df, ##y, purchase, {,...
2	track_order	change_order	[[CLS], need, help, to, co, ##rre, ##c, order,...
3	cancel_order	track_order	[[CLS], i, want, help, ol, ##cating, purchase,...
4	delete_account	create_account	[[CLS], help, me, cr, ##eti, ##ng, a, {, {, ac...
5	get_invoice	newsletter_subscription	[[CLS], i, want, to, know, about, receiving, y...
6	delivery_period	track_order	[[CLS], where, do, i, check, my, last, purchas...
7	track_order	change_order	[[CLS], i, cannot, ##ed, ##it, purchase, {, {,...
8	get_refund	newsletter_subscription	[[CLS], i, don, ##t, know, what, i, have, to, ...

Bonus Task

Try fine-tuning a different model, such as BERT, RoBERTa, or XLNet. How does the accuracy change? How does the training time change? **Hint: You can change the model name in the cell where we load the model.**

In [27]:
```
# Load a different model
model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncase
```

```
# Move everything to the GPU/MPS, if available
model.to(device)

trainer = Trainer(
    model=model,                                # the model to be trained
    args=training_args,                         # training arguments, defin
    train_dataset=tokenized_dataset['train'],   # training dataset
    eval_dataset=tokenized_dataset['test'],     # evaluation dataset
    data_collator=data_collator,                # data collator
)

trainer.train()
```

```
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at bert-base-uncased and are newly initialized: ['classifie
r.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use
it for predictions and inference.
```

[4032/4032 05:02, Epoch 3/3]

Epoch	Training Loss	Validation Loss
1	0.002200	0.018212
2	0.000700	0.008751
3	0.000500	0.009396

Out[27]:
```
TrainOutput(global_step=4032, training_loss=0.09413985109642367, metrics=
{'train_runtime': 303.045, 'train_samples_per_second': 212.81, 'train_steps
_per_second': 13.305, 'total_flos': 666944002864980.0, 'train_loss': 0.0941
3985109642367, 'epoch': 3.0})
```

In [28]:
```
# Evaluate performance

predictions = trainer.predict(tokenized_dataset['test'])
predictions = predictions.predictions.argmax(-1)
labels = tokenized_dataset['test']['label']

accuracy = accuracy_score(labels, predictions)
print(f'Accuracy: {accuracy:.2f}')
```

```
Accuracy: 1.00
```