Homework 6: Reclassifying Recipes

Due November 2nd at 10pm

Part 1: Expanding Our Network

Last week we built a regression classifier for classifying recipes into courses. We hand-selected features to feed into our model, and achieved fairly good performance. But it's annoying to have to build the features ourselves!

In this part of the assignment, you'll build a better classifier by using a neural network to extract features from the titles of the recipes.

As a human, you can probably guess the category of most of these recipes from their titles alone. But up until this point, we have not been making good use of our recipe titles. Maybe you hand-crafted features to capture some of the information in the titles last week, but you had to guess which words were important.

This week we will outsource the work of feature selection to another, more powerful network. We will take our recipe titles and turn them into feature vectors by using a powerful language model called DistilBERT, which, given a word, outputs a 768-dimension **embedding** (feature representation) of the word.

I have given you a new file to work with called **recipe_classifier.py**. You can run this file in two modes: "train", to train your model, or "interactive", as described in Task 4. To run in "train" mode, run:

```
python3 recipe_classifier.py train
```

A Note on Training Time and Model Checkpointing

We will work with large input representations this week. This will make the code much slower. **You should expect your models to take around 30 minutes to run; perhaps an hour, depending on your computer.** (If you run into difficulty, you can run your model on a lab machine. Let me know if you need help with this.)

To make things easier, I have set up the starter code to **checkpoint** your model: it saves progress checkpoints as it goes, so that if you stop and start training without losing your progress. **To retrain from scratch, you must delete these checkpoint files (found in the ckpt directory).**

Task 1: Incorporating Word Embeddings

The model that I have given you extends our regression model with DistilBERT embeddings. The **prep_bert_data** function takes the recipe titles and **tokenizes** them: translates them into lists of

indices in the DistilBERT vocabulary. This involves truncating and padding them to the same length.

For instance, the tokenizer function maps the title "Tuscan-Style Potato Soup" to the following indices:

101:	[CLS]
10722:	tu
29378:	##scan
1011:	-
2806:	style
14557:	potato
11350:	soup
102:	[SEP]
0:	[PAD]

(Notice that DistilBERT uses subword tokens in its vocabulary, so some words get split into multiple indices.)

The model that I have given you is still a regression model, but it takes these much larger, automatically-extracted vectors of features.

Because the embedding layer adds a dimension, we will need to **flatten** our input back to its original 2D shape before passing it to the Linear layer for prediction.

Paste in your regression model code into the RecipeClassifier class definition. The input to your layers is now named **flattened** rather than **inputs**.

Run the model and observe how well it does. Record its loss and accuracy.

(You will see some warnings when you run your code. This is because we are taking layers from a pretrained model, but we are freezing the weights. You can ignore these warnings.)

Task 2: Masking Padding

There is a subtle mistake in how we are feeding our data into the model. Because recipe names can be different lengths, we are truncating long ones and padding short ones. However, we haven't told the model to ignore the padding. This has a small but noticeable effect on performance.

To fix this, we can pass in an *attention mask* to tell the model which tokens to ignore. An attention mask is a list containing a 1 for each token the model should pay attention to, and a 0 for each token it should ignore.

Write a function called **get_attention_mask**. This function should take a batched item from a dataloader (i.e., like what is stored in the variable **input_ids** in the **train** function), and return a tensor containing attention masks for each item in the batch.

Recipe: "Tuscan-Style Potato Soup"
Tokens: [‘[CLS]’, ‘tu’, ‘##scan’, ‘-’, ‘style’, ‘potato’, ‘soup’, ‘[SEP]’, ‘[PAD]’]]

Token Ids: [101, 10722, 29378, 1011, 2806, 14557, 11350, 102, 0]
Attention Mask: [1, 1, 1, 1, 1, 1, 1, 1, 0]

Hint: try constructing a list of attention mask lists first, and then converting it to a tensor before returning it.

You should then add a call to **get_attention_mask** in the **train**, **predict**, and **test** functions. You should pass a tuple containing the batch and its mask as input to the *model* call.

Finally, you will need to modify your **forward** function so that it takes the attention mask as well as the input ids, and use the attention mask in your call to the embedding model:.

```
embeddings = self.distilbert(input_ids=input_ids,attention_mask=attention_mask).last_hidden_state
```

Task 3: Layering Up

We're using embeddings now, but our model is still a regression model. Now it's time to add **more parameters** to our model and make it into a neural network.

Let's add a **hidden layer** to our network. You should also add a non-linear activation: use GELU, which is a variant of ReLU. The output from these steps will be flattened and then passed in the Linear classifier layer to make a final prediction.

These layers should keep the input and output size the same as the embeddings (768).

Add a Linear layer and a GELU layer to your network before flattening. Retrain the model and observe how well it does. Record your model loss and accuracy. Do you see much of a gain from adding a hidden layer?

Task 4: Adversarial Input and Model Interpretability

When we add hidden layers, there's a trade-off in interpretability. Before, we could measure how useful each feature was to our model by looking at its weight. Now, our best tool for understanding how the model is making its decisions is to manipulate our input. This is called a **probe task**.

You can run your program in "interactive" mode to interactively probe your model with new recipe titles. To run in "interactive" mode, run:

```
python3 bert_model.py interactive
```

Play around with this, and try to find some **minimal pairs**: pairs of recipe titles that are almost the same, but are classified differently by the model.

Construct 3 minimal pairs that illustrate aspects of how the model makes its decisions. For each example, make a hypothesis about which features in the input the model is paying attention to, and discuss how your minimal pair supports this hypothesis. At least one of your examples should illustrate a **model error**: a case where a human can guess the correct category, but the model cannot.

Task 5: Model Generalizability

I have also given you a dataset of medieval recipes (**medieval_recipes.csv**, scraped from medieval-cookery.com). Test out your recipe classifier on this dataset.

How well does it do? What kind of mistakes to you observe?

Part 2: Reading Response

Question 1 (5 points)

"Purpose and Profit" covers the development of the Internet. **What surprised you about this history?**

Question 2 (10 points)

A major question in this chapter is how decisions should be made about an open public utility like the Internet. For instance, David Clark is quoted as saying that the early internet pioneers "reject[ed] presidents, kings, and voting", but believe in "rough consensus." The chapter also discusses how competition impacted these decisions once the Internet was privatized.

Discuss the advantages and disadvantages of making decisions about technology through voting, consensus-building (i.e. through public discussion), and commercial competition.